US006389560B1

(12) **United States Patent**
Chew

(10) **Patent No.: US 6,389,560 B1**
(45) **Date of Patent: *May 14, 2002**

(54) **UNIVERSAL SERIAL BUS INTERPRETER**

(75) Inventor: **Michael N. Chew, San Jose, CA (US)**

(73) Assignee: **Sun Microsystems, Inc., Palo Alto, CA (US)**

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

This patent is subject to a terminal disclaimer.

(21) Appl. No.: **09/232,983**

(22) Filed: **Jan. 19, 1999**

(51) **Int. Cl.**[7] **H02H 3/05**
(52) **U.S. Cl.** **714/43; 714/44; 710/99**
(58) **Field of Search** 714/40, 43, 44, 714/56; 710/17, 19

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,859,993	A	*	1/1999	Snyder	712/208
5,974,486	A	*	10/1999	Siddappa	710/53
6,012,103	A		1/2000	Sartore et al.	
6,044,428	A	*	3/2000	Rayabhari	710/129
6,098,120	A		8/2000	Yaotani	
6,101,076	A	*	8/2000	Tsai	361/90
6,105,097	A		8/2000	Larky et al.	
6,119,194	A		9/2000	Miranda et al.	
6,157,975	A	*	12/2000	Brief	710/104
6,170,062	B1		1/2001	Henric	
6,178,514	B1		1/2001	Wood	
6,185,569	B1		2/2001	East et al.	
6,202,103	B1	*	3/2001	Vonbank	710/15
6,219,736	B1	*	4/2001	Klingman	710/129

FOREIGN PATENT DOCUMENTS

JP	11-87597	*	10/2000

OTHER PUBLICATIONS

Borriello, Gaetano et al. "Interface Synthesis: a vertical slice from didgital to software components", ACM 1998 1-58113-008 Feb. 1998.*

Breevoort, C.M. "A Multi-services Communications Architecture For In-home USB and IEEE-1394 Based Devices" IEEE 1998.*

Adams et al, "Conformance Testing of VMEbus and Multibus II Products," IEEE Micro, vol. 12, No. 1, Feb. 1992, pp. 57-64.

Zelms, "On-Line Diagnosis of Peripherals in a Minicomputer System," Proceedings of the National Electronics Conference, Oct. 1980, pp. 545-550.

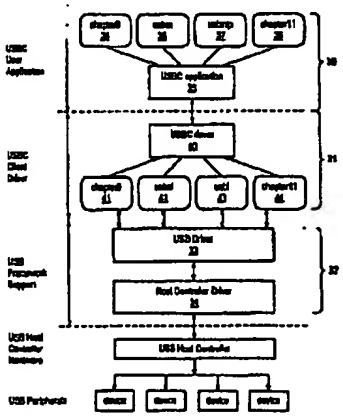
(List continued on next page.)

Primary Examiner—Robert Beausoleil
Assistant Examiner—Bryce P. Bonzo
(74) *Attorney, Agent, or Firm*—Conley, Rose & Tayon, PC; B. Noël Kivlin

(57) **ABSTRACT**

A system and method for testing the conformance of a universal serial bus (USB) system to a set of predefined USB Specifications. One embodiment of the system comprises a USB interpreter that can be used to selectively examine device data, execute USB commands and exercise USB functions without having to create or compile a test program. The USB interpreter comprises a test application and a test application driver. The test application driver interfaces with the USB system software. The USB system software may include a USB driver, a host controller driver and other host software. The USB driver interfaces with the test application through the test application driver. The host controller driver interfaces with the host controller and thereby interfaces the software on the host system with the USB interconnect and USB devices. In one embodiment, the USB interpreter incorporates a command line interpreter through which a user can enter commands to perform specific operations and tests on the USB system. The user may execute commands in an operating system (e.g., Unix) shell without having to interrupt a USB testing or debugging session. The user may also enter commands and perform USB system testing remotely via a communications link between the user and the system's host computer.

29 Claims, 6 Drawing Sheets



OTHER PUBLICATIONS

Universal Serial Bus Specification Revision 1.0, Jan. 1996, pp. 2–9, 165–172.
Dreitlein, "The Challenge of Testing SCSI Peripherals," Electronics Test, vol. 13, No. 6, Jun. 1990, 4 pgs.
European Search Report, Application No. 00 30 0324, mailed Jul. 26, 2000.
"Universal Host Controller Interface (UHCI) Design Guide; Revision 1.1", Intel Corporation, Mar. 1996.

* cited by examiner

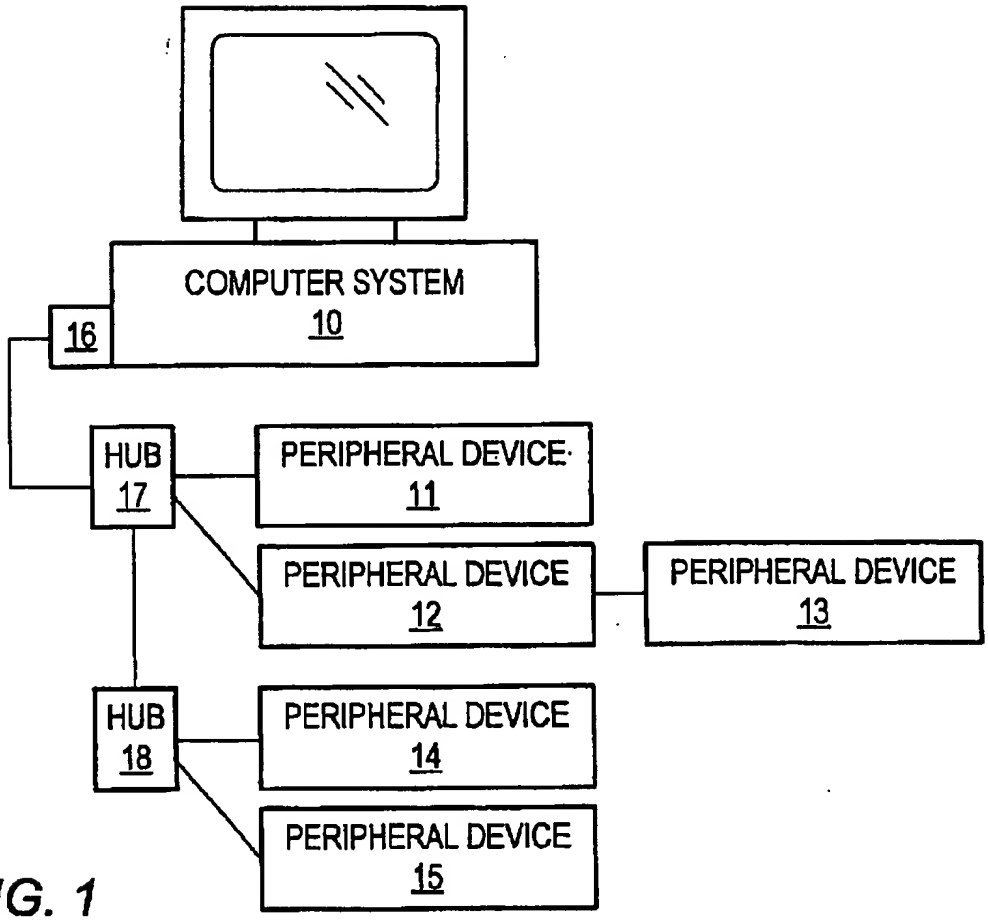

COMPUTER SYSTEM
10
16
HUB
17
PERIPHERAL DEVICE
11
PERIPHERAL DEVICE
12
PERIPHERAL DEVICE
13
HUB
18
PERIPHERAL DEVICE
14
PERIPHERAL DEVICE
15

FIG. 1

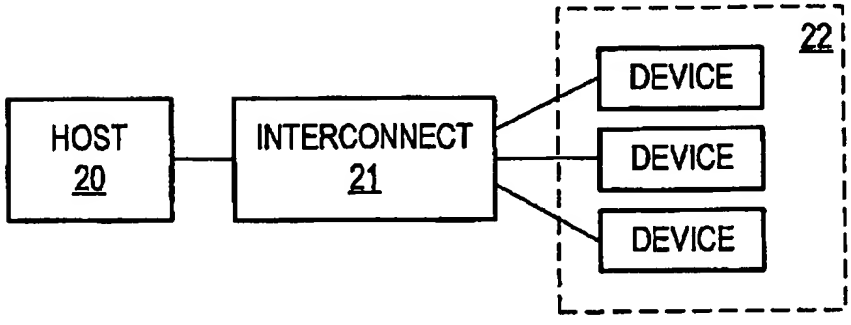

HOST
20
INTERCONNECT
21
DEVICE
DEVICE
DEVICE
22

FIG. 2

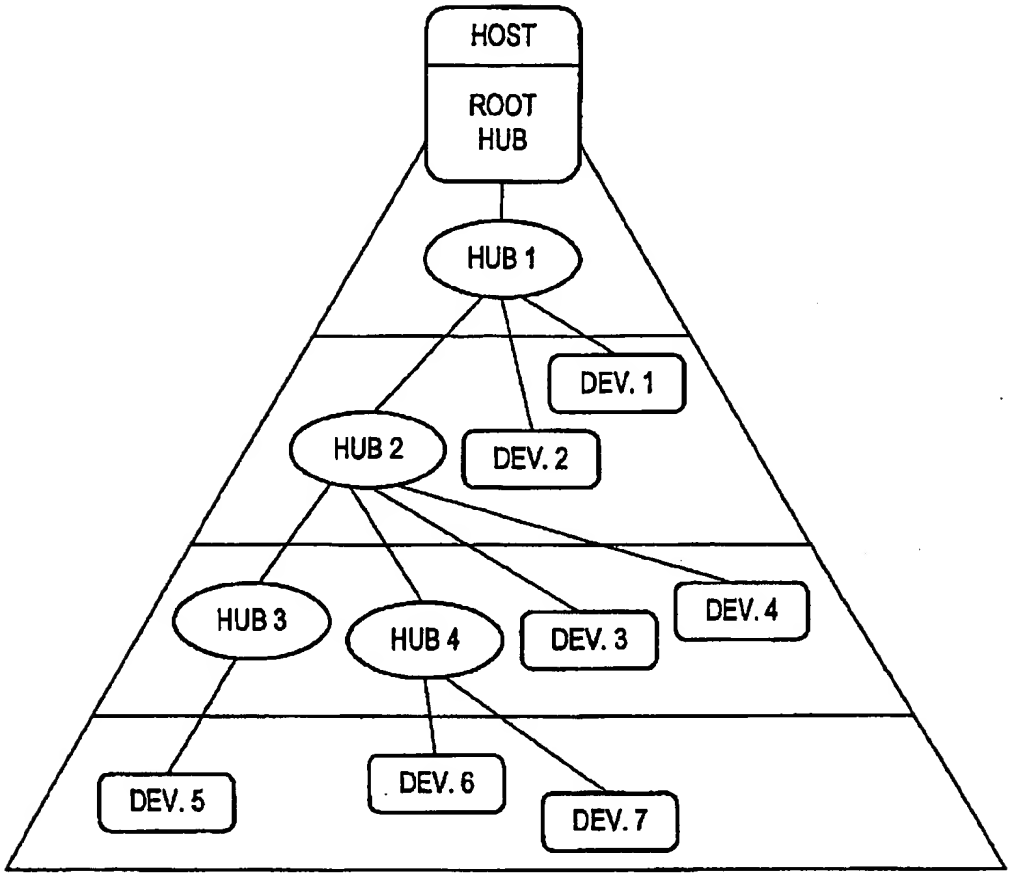

HOST
ROOT HUB
HUB 1
DEV. 1
HUB 2
DEV. 2
HUB 3
HUB 4
DEV. 3
DEV. 4
DEV. 5
DEV. 6
DEV. 7

FIG. 3

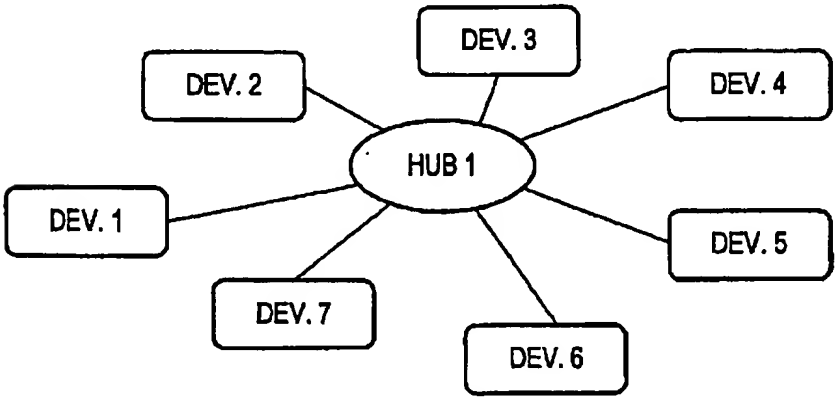

DEV. 3
DEV. 2
DEV. 4
HUB 1
DEV. 1
DEV. 5
DEV. 7
DEV. 6

FIG. 4

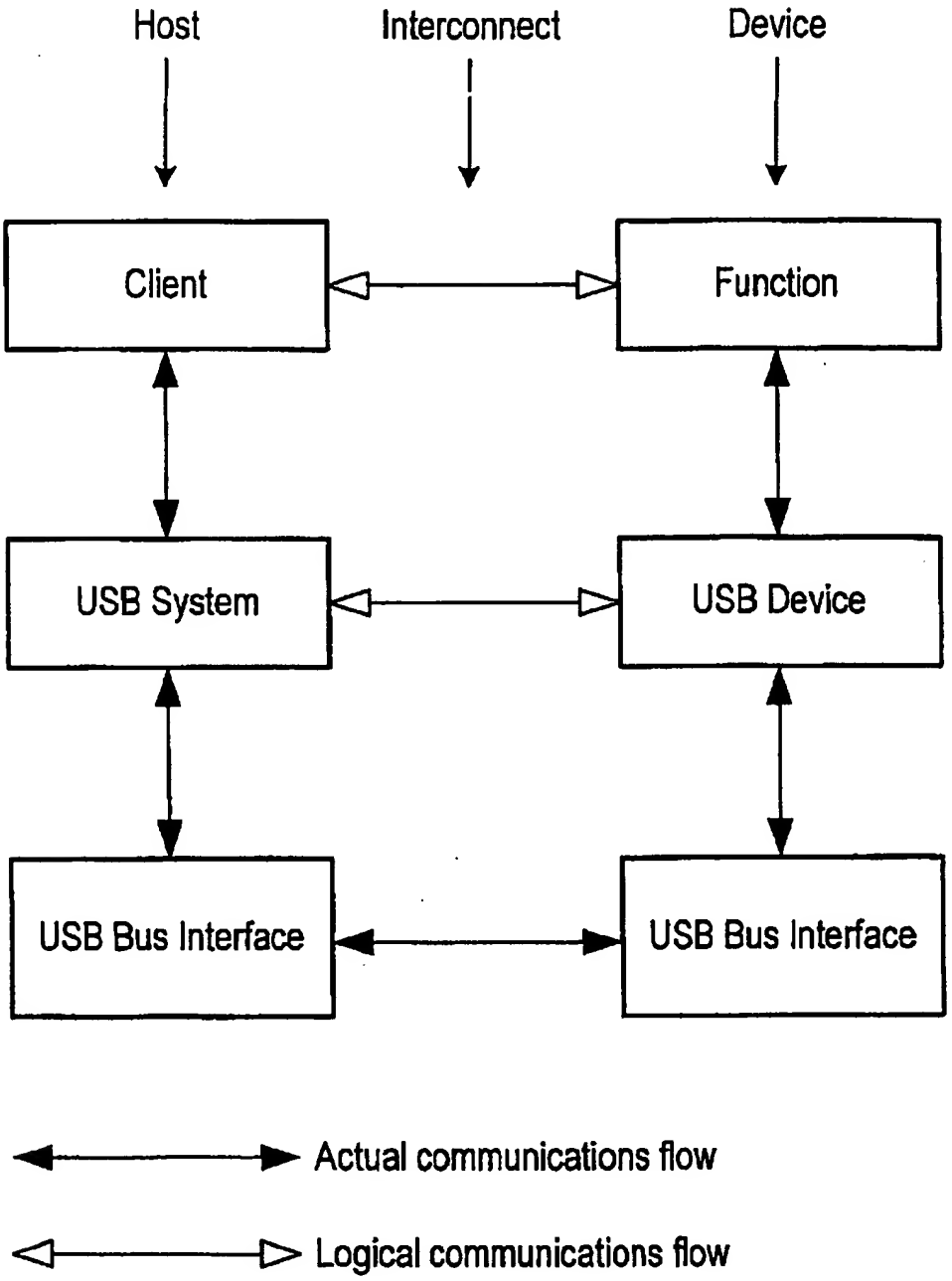

Host
Interconnect
Device
Client
Function
USB System
USB Device
USB Bus Interface
USB Bus Interface
Actual communications flow
Logical communications flow

FIG. 5

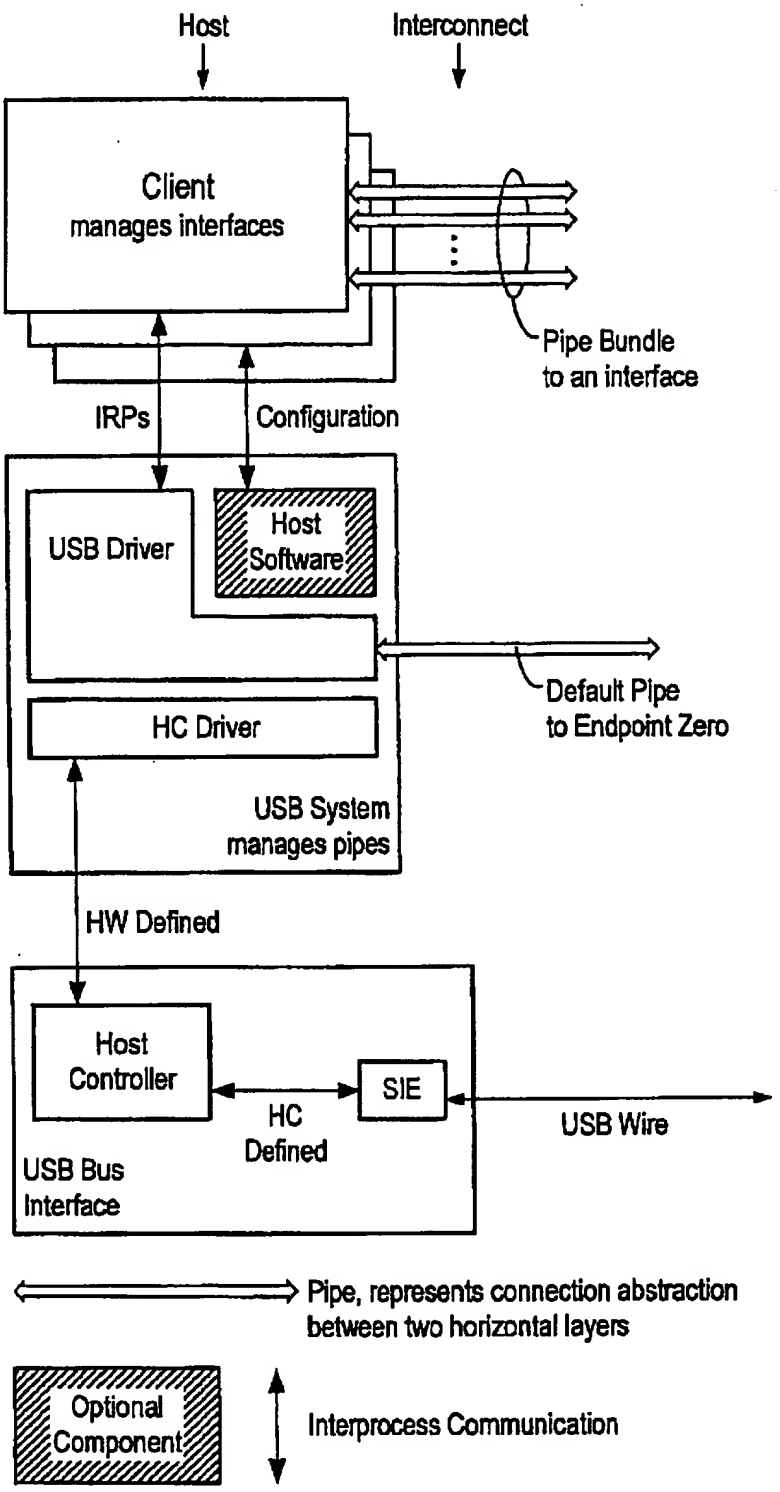

Host
Interconnect
Client
manages interfaces
Pipe Bundle
to an interface
IRPs
Configuration
USB Driver
Host
Software
Default Pipe
to Endpoint Zero
HC Driver
USB System
manages pipes
HW Defined
Host
Controller
SIE
HC
Defined
USB Wire
USB Bus
Interface
Pipe, represents connection abstraction
between two horizontal layers
Optional
Component
Interprocess Communication

FIG. 6

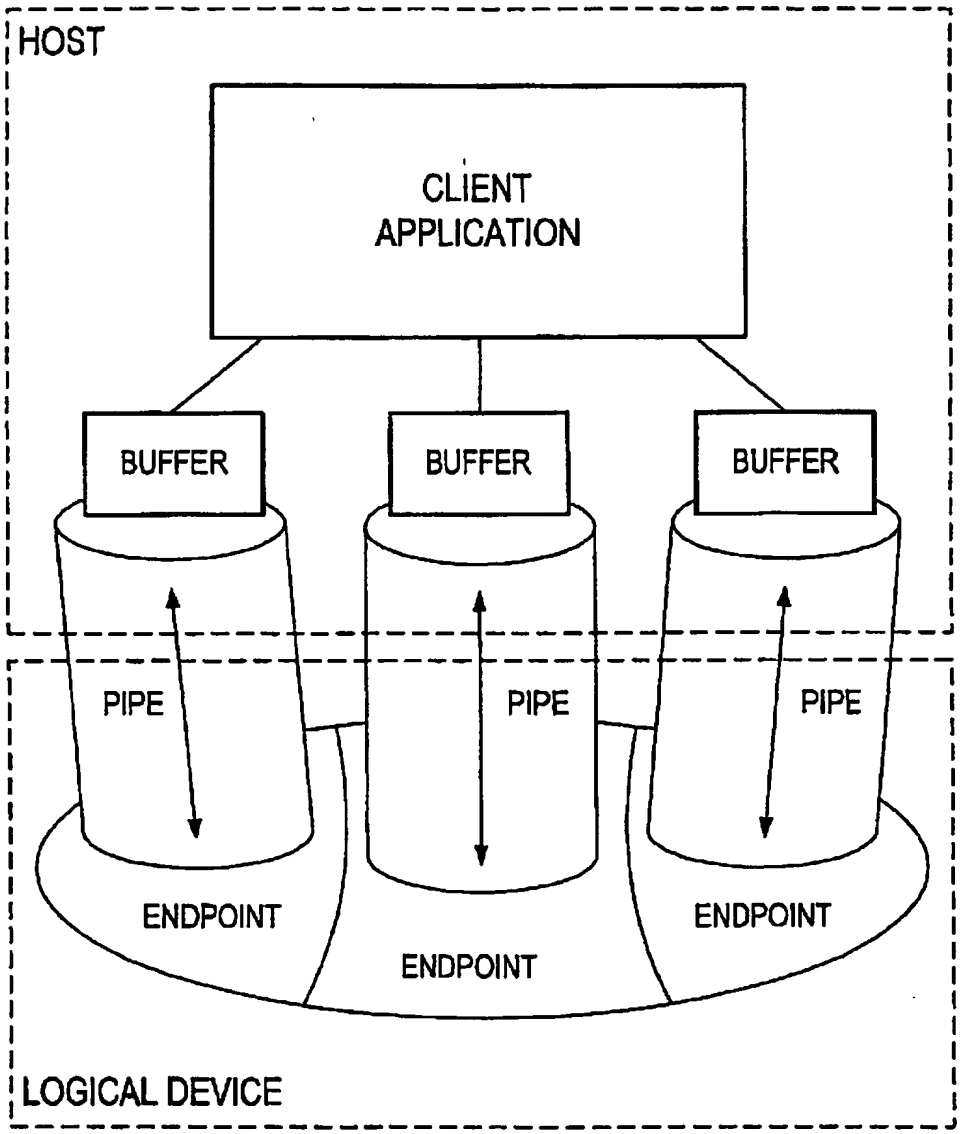

HOST
CLIENT APPLICATION
BUFFER
BUFFER
BUFFER
PIPE
PIPE
PIPE
ENDPOINT
ENDPOINT
ENDPOINT
LOGICAL DEVICE

FIG. 7

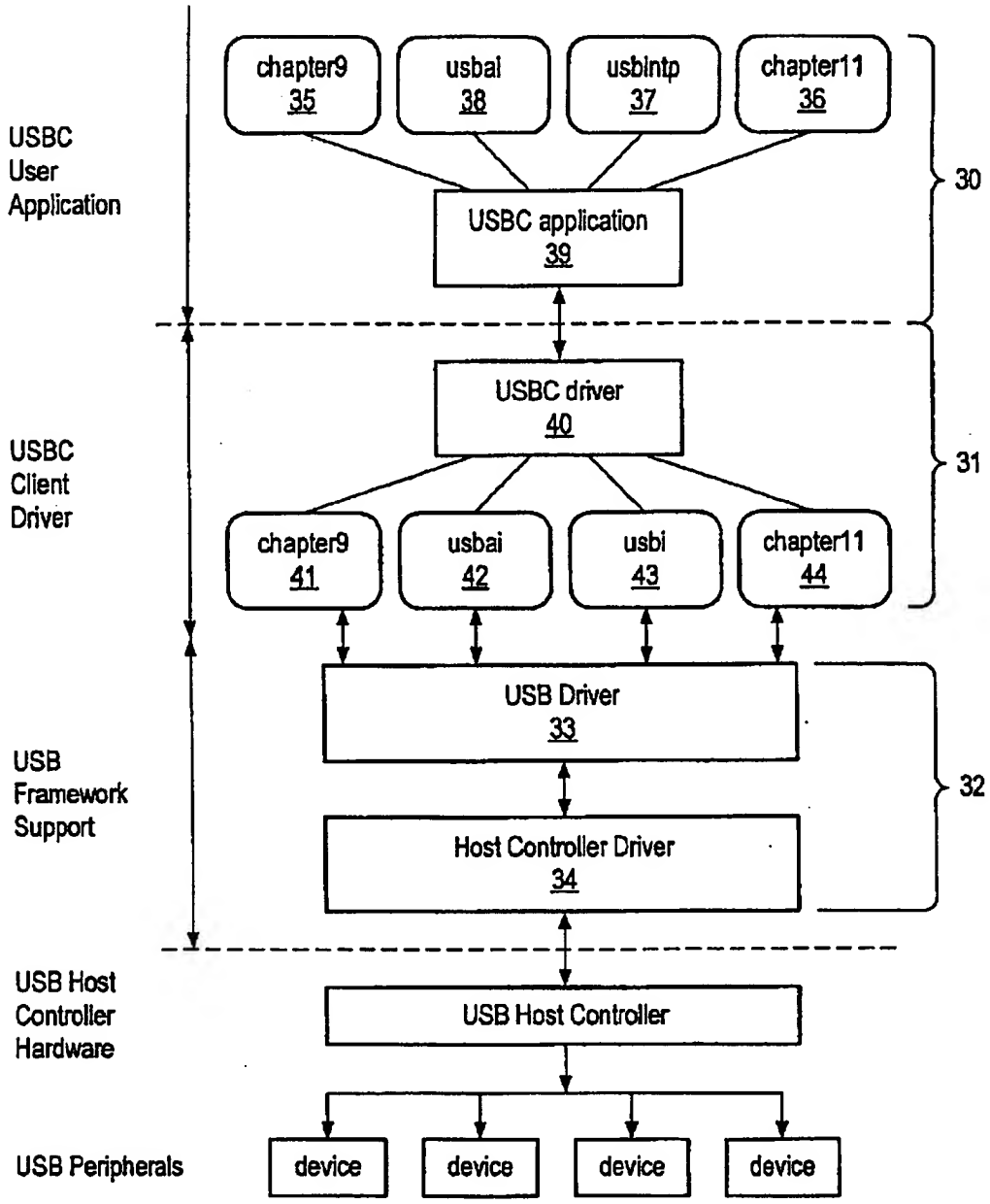

chapter9
35
usbai
38
usbintp
37
chapter11
36
USBC
User
Application
USBC application
39
30
USBC driver
40
USBC
Client
Driver
chapter9
41
usbai
42
usbi
43
chapter11
44
31
USB Driver
33
USB
Framework
Support
Host Controller Driver
34
32
USB Host
Controller
Hardware
USB Host Controller
USB Peripherals
device
device
device
device

FIG. 8

UNIVERSAL SERIAL BUS INTERPRETER

BACKGROUND OF THE INVENTION

1. Field of the Invention

The invention relates generally to computer systems and more particularly to methods and devices for testing the functional compatibility of peripheral devices with a universal serial bus.

2. Description of the Relevant Art

Since the advent of personal computers, computer users have been eager to expand the capabilities of their machines. Users, however, have experienced innumerable difficulties when confronted with the task of connecting peripheral devices to their computers. While it may be simple for a user to attach a printer to his or her computer, the connection of a device (e.g., a scanner) to a serial port presents more of a challenge. The installation of equipment internal to the computer, such as an interface card for a scanner, may present even greater difficulties, as the user may face problems in setting I/O and DMA addresses for resolving IRQ conflicts. These difficulties can frustrate the user, particularly when they cause the computer to operate incorrectly or simply fail to operate at all.

With the rapid advances in the state of computer technology, the potential for experiencing such difficulties has grown. There have, as a result, been attempts to alleviate these problems. For example, the concept of designing plug-and-play peripheral devices was intended to alleviate difficulties of installing the devices. This concept, however, is directed primarily toward devices which are installed inside the cabinet of the computer. The installation of external peripheral devices, such as printers and scanners, is still likely to be accompanied by some of the difficulties targeted by the plug-and-play concept.

Another attempt to eliminate some of the problems attendant to the installation of peripheral devices was the introduction of PC-Card technology. (This technology was formerly termed PCMCIA—Personal Computer Memory Card International Association.) PC-Card (PCMCIA) peripheral devices are simply and easily inserted into a PC-Card socket and are recognized by the computer. The problem with this technology, however, was that it was originally targeted to portable computers. Although a PC-Card (PCMCIA) slot can be installed in a desktop computer, this solution simply has not been widely adopted. Thus, there remained a need for a simple and convenient plug-and-play type technology for desktop computers.

SUMMARY OF THE INVENTION

One or more of the problems outlined above may be solved by various embodiments of the system and method of the present invention. In response to the continuing difficulties in installing peripheral devices and the need for a solution to the problem, the idea of a universal serial bus (USB) was developed. The development of the USB was motivated by number of factors, including the difficulty of adding peripheral devices and the lack of additional ports for installing these devices. The USB is designed to provide plug-and-play capabilities for external peripheral devices which are connected to the I/O ports of the computer and thereby reduce the difficulties experienced by many users. The USB was also designed to provide means for installing numerous devices rather than restricting the user to one or two (one for each port on a computer which does not have a USB).

The implementation of plug-and-play capabilities through the USB is not solely dependent upon the USB. It is fundamental that the peripheral devices to be installed on the USB must be compatible with the USB. In other words, it is necessary that the devices conform to the specific characteristics of the USB. This is ensured in part by the propagation of the USB Specification, which defines these characteristics. The USB Specification is hereby incorporated herein by reference in its entirety. The designs of peripheral devices can be checked prior to manufacture through device simulations. Such verification of device designs, however, may themselves contain errors. Additionally, errors may be introduced in translation of the design into a physical device. It is therefore important to have means for verifying different aspects of USB compatibility of peripheral devices in their final physical configurations. It is also important to have means for verifying USB system functions apart from the peripheral devices. The various embodiments of the invention provide such means.

One embodiment of the invention comprises a USB interpreter. The USB interpreter is a software tool that can be used in a USB system to selectively examine device data, execute USB commands and exercise USB functions. The USB interpreter can perform these functions without having to create or compile a test program and can therefore be very useful in debugging devices with respect to USB compliance. The USB interpreter can also be used in the development of USB software.

The USB interpreter comprises a test application and a test application driver. The test application driver interfaces with the USB system software. The USB system software, which may include a USB driver, a host controller driver and other host software, is sometimes referred to as the USB framework support. The USB driver interfaces with the test application through the test application driver. The host controller driver interfaces with the host controller, which in turn interfaces the software on the host system with the USB interconnect and USB devices.

In one embodiment, the USB interpreter incorporates a command line interpreter through which a user can enter commands to perform specific operations and tests on the USB system. The user may thereby avoid performing unnecessary tests on previously verified portions of the system. The use of the command line interpreter further allows the user to execute commands in an operating system (e.g., Unix) shell without having to interrupt a USB testing or debugging session. The use of the command line interpreter also allows the user to enter commands remotely (e.g., via a modem connected to the computer system) so that the expertise of a user who is not located at the site of the computer system.

BRIEF DESCRIPTION OF THE DRAWINGS

Other objects and advantages of the invention will become apparent upon reading the following detailed description and upon reference to the accompanying drawings in which:

FIG. 1 is a block diagram illustrating the physical configuration of a plurality of USB peripheral devices attached to a computer system.

FIG. 2 is a block diagram illustrating the primary physical components of a USB system.

FIG. 3 is a block diagram illustrating the tiered star physical configuration of a plurality of devices connected to hubs on a USB interconnect.

FIG. 4 is a block diagram illustrating the simple star logical configuration of a plurality of devices connected to a USB interconnect.

FIG. 5 is a functional block diagram illustrating the logical and physical flows of data within a USB system.

FIG. 6 is a functional block diagram illustrating the logical and physical flows of data within the host in a USB system.

FIG. 7 is a diagram illustrating the flow of data between client software in the USB host and a plurality of endpoints in a USB device.

FIG. 8 is a diagram illustrating the structure of the USB test application in one embodiment of the invention.

While the invention is susceptible to various modifications and alternative forms, specific embodiments thereof are shown by way of example in the drawings and will herein be described in detail. It should be understood, however, that the drawing and detailed description thereto are not intended to limit the invention to the particular form disclosed, but on the contrary, the intention is to cover all modifications, equivalents and alternatives falling within the spirit and scope of the present invention as defined by the appended claims.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

One embodiment of the invention is described below. In this embodiment, a host computer utilizes a USB system. The USB system includes a USB, a USB host controller coupled to the USB, a host controller driver for driving the host controller and a set of USB interfaces which allow communications between a test application and the host controller driver. The test application comprises a USB interpreter which is used to selectively test the functions of the USB system. Specifically, the application is configured to examine standard device descriptors, query USB devices using standard device requests and exercise individual USB interface functions. The USB interpreter allows the user to take these actions without having to first create and compile a test application. The USB interpreter thereby allows a user to selectively obtain information which facilitates debugging of USB devices and related software.

The development of the USB was motivated primarily by three considerations. First, personal computers have traditionally had limited flexibility in regard to reconfiguration of the computer. A number of advances were made in the areas of graphical user interfaces and internal bus architectures which made personal computers more user-friendly, but there was little progress in improving the connectivity of peripheral devices to desktop systems (despite the success of PC-Card plug-and-play peripherals in portable computers.) Second, personal computers typically had a limited number of ports to which peripheral devices could be connected. A typical system, for example, might have a single parallel port and one or two serial ports. Users were therefore prevented from having more than two or three peripheral devices corresponding to the two or three ports on their computers. Third, although there has been significant potential for computing and communication functions to benefit from each other, these technologies have evolved essentially independently so that the technologies were not easily merged. There was therefore a need for an easy and inexpensive means to communicate information via computers. The USB was designed to meet these needs.

Referring to FIG. 1, the USB is a bus designed to provide a simple and efficient method for connecting external peripheral devices to desktop computer systems. The figure shows a computer system **10** connected to several peripheral devices **11–15**. The devices are connected to the USB port **16** on the computer system via hubs **17–18**. The use of hubs **17–18** on the USB enables users to expand the number of devices which can be connected to the computer system (as compared to the two or three which could be directly connected to a non-USB system.) The system may also include compound devices **12** which serve as both hubs and functional devices. (Note that device **13** is connected to the USB via compound device **12**.)

Referring to FIG. 2, a block diagram illustrating the primary physical components of a USB system is shown. The physical USB system can be described in three parts; a USB host **20**; USB devices **22**; and a USB interconnect **21**. The first of these parts, the USB host, is the computer system which incorporates the USB root hub and forms the basis of the USB system. The second part, the USB devices, comprise the peripherals and functional devices which are to be connected to the computer system. USB "devices" may also refer to the hubs which can be connected to the USB to provide additional attachment ports. The third part, the USB interconnect, comprises the physical connections between the USB devices and the USB host, as well as the manner in which the devices communicate with host.

There is a single host associated with any USB system. The host is the computer system in which the USB system is implemented. The host incorporates a root hub of the USB which provides one or more attachment points for devices or other hubs. A host controller provides the interface between the host and the USB. The host controller may be implemented in one or a combination of hardware, firmware and software.

The USB devices are functional devices which provide capabilities to the system (e.g., an ISDN modem, a joystick or a set of speakers.) The USB devices may also be hubs which provide additional attachment points to the USB to which additional devices may be connected. (Non-hub devices are sometimes referred to as functions.) Some USB devices serve as both functional devices and hubs to which other devices can be attached. The USB Specification requires that all USB devices conform to certain interface standards and thereby ensures that the devices comprehend the USB protocol and respond to standard USB requests and commands.

The physical aspects of the USB interconnect are defined by the bus topology. (Although the bus topology includes non-physical aspects of the USB interconnect, they will be addressed elsewhere in this disclosure.) The bus topology describes the manner in which the USB connects USB devices with the USB host. Referring to FIG. 3, the physical configuration of the USB interconnect is that of a tiered star. The host has a root hub which forms the basis of the interconnect. Devices and/or additional hubs can be connected to the root and other hubs to form successive tiers of the interconnect. Thus, each hub forms the center of one of the stars in the tiered star configuration. Each wire segment in the interconnect is a point-to-point connection between the host or a hub and another hub or a device.

USB systems support "hot plugging". That is, USB devices may be attached to or removed from the USB at any time. The USB is designed to detect these changes in its physical topology and accommodate the changes in the available functions. All USB devices are connected to the USB at one of the hubs (either the root hub or one of the hubs chained from the root.) Attachment or removal of a device at a hub is indicated in the hub's port status. If a device is attached to the hub, the hub sends a notification to the host. The host then sends a query to the hub to determine

the reason the notification was sent to the host. In response to this query, the hub sends the number of the port to which the device was attached to the host. The host then enables this port and begins communicating with the device via the control pipe (0 endpoint.) The host determines whether the attached device is a hub or a function and assigns a unique USB address to the device. The unique USB address and the 0 endpoint of the device are used as a control pipe for the device. If the newly attached device is a hub which already has devices attached to its ports, this same process is repeated for each of the attached devices. After a device has been attached and communications established between the device and the host, notifications are sent to interested host software.

If a device is removed from a hub, the hub disables the port to which the device had been attached and sends notification of the device's removal to the host. The host then removes the device and related data from any affected data structures. If the removed device is a hub to which other devices are attached, the removal process is repeated for each of the devices attached to the removed hub. Notifications are sent to interested host software indicating that the removed devices are no longer available.

Although the physical topology of the USB interconnect is that of a tiered star configuration, the logical topology of the system is a simple star as shown in FIG. 4. Alternately, the logical configuration can be considered a series of direct connections between the individual USB devices and a client software application on the host. The logical relationship of the client software to the devices can also be thought of as one or more direct connections between the client software and the specific functions provided by the devices. While the view of the logical configuration as being a series of direct connections holds true for most operations, the system remains aware of the tiered physical topology so that devices downstream from a removed hub can be removed from the logical configuration when the hub is removed (see the discussion of hot-swapping above.)

The USB provides means for communications between client software running on the host and functions provided by the USB devices. FIGS. 5 and 6 illustrate the flow of data which is communicated between the client software and the device functions. The figures show the host as comprising three components: the client software; the USB system software (including USB driver, host controller driver and host of software); and the USB host controller. The host controller driver interfaces the host controller with the USB system software, and the USB driver interfaces the USB system software with the client software. FIG. 5 shows that the USB device also comprises three components: the function; the logical device; and the USB bus interface.

While the logical flow of information between the client software and the function is direct, the figure shows that the actual flow of data goes from the client software to the USB system software, to the USB host controller, to the USB bus interface, to the USB logical device and finally to the function. Likewise, although the logical flow of control information from the USB system software to the USB logical device is direct, the actual flow of information must go through the host controller and the bus interface.

Referring to FIG. 7, a USB logical device is viewed by the USB system as an interface formed by a collection of endpoints. An endpoint is a uniquely identifiable portion of a USB device that forms the end of a communications path from the host to the device. Software may only communicate with a USB device via its endpoints. (The communications flow is illustrated in the figure by the arrows.) The number of each endpoint is determined by the designer of the device. The combination of a device address (assigned by the system at device attachment time) and the endpoint number allows each endpoint to be uniquely identified.

All USB devices are required to have an endpoint with number 0. This endpoint is used to initialize and manipulate (e.g., to configure) the logical device. Endpoint 0 provides access to the device's configuration information and allows access to the device for status and control purposes. Devices can have additional endpoints as required to implement their functions. Devices can have up to 16 additional input endpoints and 16 additional output endpoints (unless they are not full-speed devices, in which case they are limited to a reduced number of endpoints.).

The communications path between an endpoint on a device and software on the host is referred to as a pipe. A pipe comes into existence when a USB device is configured. Software clients normally request data transfers via I/O Request Packets (IRPs) to a pipe. The software clients then either wait or are notified when the requests are completed. Endpoint 0 has an associated pipe called the Default Pipe. The Default Pipe is used by system software to determine device identification and configuration requirements and to configure the device. The Default Pipe can also be used by device specific software after the device is configured, but the USB system software retains "ownership" of the Default Pipe and controls its use by client software.

Referring to FIG. 8, the software structure of one embodiment of the invention is shown. The client software in this embodiment comprises a test application 30 and a test application driver 31. The USB system software 32 comprises USB driver 33 and host control driver 34. The USB driver 33 and host control driver 34 are part of the USB Framework Support. The USB Framework Support is a Solaris® based implementation of the USB system software and includes a set of interfaces (USB Architecture Interfaces, or USBAI) which allow third party vendors to write USB client drivers on a Solaris® SPARC® platform.

Test application 30 includes modules configured to control testing of the different functions of the USB system. In this instance, the separation of the modules' capabilities generally conforms to the separation of USB functionalities defined in the USB Specification. For example, one module 35 controls the testing of standard device requests defined in Chapter 9 of the Specification, while another module 36 controls testing of hub standard requests as defined in Chapter 11 and another module 38 controls the testing of USBAI functions. Interpreter module 37 does not provide for the testing of a separate set of functions, but instead supports testing of all of the USB functions. These modules are all operatively coupled to the body 39 of the test application.

Test application driver 31 is similarly structured, having a main driver component 40 and several modules 41–44 which correspond to the modules of application 30. Test application driver 31 is a loadable driver. That is, when a USB device is hot-plugged, the USB architecture framework will load the driver and create a device node for the newly installed device. If the device is hot-unplugged, the driver will be unloaded as to the unplugged device.

The test application may be run on any USB device after it is installed. In one embodiment, the application, which is controlled primarily by the interpreter module, takes a device node as an argument, opens the device node and constructs state information for the device based on descrip-

tor information obtained from the device. Test application driver 31 maintains the device state for use in testing the device. System test resources are allocated based on the constructed state information. The state information is also used as the basis of test requests which may be formulated by application 30. The test requests are forwarded to USBI module 43 of the test application driver 31, which decodes and validates the test parameters. If the test parameters are valid (i.e., within the allowable limits of the parameters,) the test application 30 and application driver 31 construct test cases using USBAI functions. The corresponding function calls are transmitted to the USB driver, which executes the functions. Test application driver 31 updates the state information for the device when the function calls are issued and notifies test application 30 of the pass/fail status of the tests after the test functions are performed. Test application 30 then notifies the user.

In one embodiment, the test application runs a suite of tests to verify all of the USBAI function calls. The application opens a device node, constructs state information for the device and allocates system resources based on the state information. The USBAI module 38 of the test application 30 formulates test requests and parameters based on the state information and conveys the test requests and parameters to the USBAI module 42 of the test application driver 31 for validation. After validating the test requests and parameters, the USBAI module of the test application driver formulates a series of test cases using the USBAI functions. The USBAI module of the test application driver then makes USBAI function calls corresponding to the test cases to the USB system software. The USBAI module of the test application driver returns a pass or fail indication to the test application after analyzing the results of the function calls. This same process is repeated for all of the endpoints in the selected USB device. The test application may spawn multiple threads to allow concurrent testing of the different endpoints. Some embodiments may also provide for concurrent testing of multiple USB devices.

In a similar manner, the test application may run a suite of tests to verify that a USB device can provide appropriate device information in response to all of the standard device requests defined in the USB Specification. The application again opens a device node, constructs state information for the device and allocates system resources based on the state information. All of the standard device requests are packaged in a request structure which is passed to the test application driver. The Chapter 9 module 41 of the test application driver 31 validates the commands in the request structure and then conveys the requests to the USB device via the Default Pipe. The information provided by the device in response to the standard device requests is then returned to the test application.

Different embodiments of the invention may include various features. One such feature is a command line mode. In command line mode, a user may input individual commands which are interpreted and executed by the application to test particular functions of the USB system. This can eliminate unnecessary testing which would be performed by a comprehensive suite of tests. Commands may, however, also be set up to execute a series of operations instead of a single operation.

The command line mode also allows the test application to be used remotely. In other words, the user does not have to be physically present to test the USB system. The user may instead establish communications with the test system and enter commands through the communications link. For example, the user may establish a modem connection between a remote computer and the test system and then enter commands via the modem connection. The link may utilize any suitable means for communicating, and the foregoing example of a modem-based link is intended to be illustrative rather than limiting.

The user may also execute commands which are unrelated to the test system (e.g., Unix shell commands) without having to interrupt the test session. The time normally required to start and terminate other test systems to perform non-system functions may therefore be avoided. The command line mode can be configured to alias the available commands to a unique list to reduce the amount of typing which is required. The test system can also be configured to provide online help to facilitate the user's interaction with the system.

One embodiment of the test system is configured with a functional mode in which the system can single-step through a USBAI function. This may be useful when the user needs to examine traffic on the USB. After a USBAI function command is issued, USB traffic may be examined using a logic analyzer as each step of the function is performed. This can be a valuable debugging feature.

Another feature which may be included is the capability of switching ports during testing. As mentioned above, the system is configured to perform device enumeration. That is, when there are multiple USB devices connected to the system, it can list all the connected devices and construct state information such as port number, device type, device class and path for each of the devices. The user can select the device at a particular port for certain tests and then switch to a different port and test the device connected to that port.

In addition to determining device information for the purpose of testing the devices, the test system may be configured to make this information available to the user. Information such as device descriptors can be displayed in a matrix format to enable the user to make side-by-side comparisons of the characteristics of individual devices. Because bus enumeration is an ongoing activity in the USB system, devices are recognized as they are connected to the USB and the information which is normally obtained on the devices can be displayed alongside information for previously installed devices. Likewise, information corresponding to devices which are removed from the USB system can be removed from the display.

In one embodiment, the commands which can be input to the test system can be grouped into four categories: commands relating to device state information; standard device request commands; USB architecture interface commands; and miscellaneous commands. Although a number of these commands are listed below, this list is not intended to be limiting. It is contemplated that the USB Specification may be amended to add, delete or modify the allowed commands to accommodate the changing functionality of the USB, and the test system may be adapted to include the new commands and functions of the USB.

The device state information commands are shown in Table 1 along with their corresponding functions. Device state information is tracked for the devices enumerated and identified by the test system. When an endpoint of a device is accessed, its state is updated in the test system.

TABLE 1

Status	Lists port number, device class and device node name for all available devices. (Usually used after the "enumerate" command.)
Enumerate	Probes all available USB device nodes and creates device state information for each of the connected USB devices.

TABLE 1-continued

Device_state	
Port	Used to switch from a port at which one device is attached to another port at which a second device is connected. (The port numbers can be obtained from the "status" command.)

The standard device requests are defined in Chapter 9 of the USB Specification. The standard device requests are shown in Table 2 along with their corresponding functions. All USB devices are required to respond to standard device requests from the host. These requests are made via the device's default pipe using control transfers. The request and the request's parameters are sent to the device in the setup packet.

TABLE 2

Get_status	Used to obtain status for a device, interface or endpoint. Device status consists of a remote_wakeup value corresponding to either enable or disable. The returned status in the interface field must be zero. The returned status of an endpoint can be either stalled or not stalled.
clear_feature	Used to clear or disable two feature selectors: DEVICE_REMOTE_WAKEUP for the device; or ENDPOINT_STALL on a specific endpoint address. The endpoint address can be obtained from the "get_descriptor" or "device_state" commands.
set_feature	Used to set or enable two feature selectors: DEVICE_REMOTE_WAKEUP for the device; or ENDPOINT_STALL on a specific endpoint address. The endpoint address can be obtained from the "get_descriptor" or "device_state" commands.
set_address	Used only by system software.
get_descriptor	Used to return the descriptors for device, configuration, string or hub descriptors. All devices must provide a device descriptor and at least one configuration descriptor. The command returns the hub descriptor only in the device is in the hub class.
set_descriptor	USB devices usually do not support this command. The stall condition should be returned.
get_config	Returns the current configuration value. If the returned value is 0, the device is not configured. If the device is configured, a non-0 configuration value is returned.
set_config	Used to set the configuration value of the configuration descriptor.
get_interface	Used by the host to determine the currently selected alternate setting.
set_interface	Used by the host to set a selected alternate setting of an interface.
synch_frame	Used only for isochronous devices. This command should generate a stall condition.

The USBAI commands are shown in Table 3 along with their corresponding functions. The USBAI commands are issued to a particular endpoint of a selected device. The USBAI functions corresponding to these commands can be executed in single-step mode in order to allow the user to examine traffic on the USB.

TABLE 3

open_pipe	Used to open individual endpoints of USB devices. The open_pipe command takes an endpoint index as an argument for opening the pipe. The endpoint index can be obtained from the "device_state" command.
close_pipe	Used to close individual endpoints. This command takes an endpoint index as an argument for closing the pipe. The endpoint must be opened before it can be closed. The endpoint index can be obtained from the "device_state" command.
start_polling	This command applies only to an interrupt endpoint. It is invalid if the argument is a non-interrupt endpoint. Before an interrupt endpoint can be polled, it must be opened using the "open-pipe" command. The endpoint index can be obtained from the "device_state" command. Most USB devices need some hardware event to generate an interrupt packet after the endpoint is polled.
stop_polling	This command applies only to an interrupt endpoint. Polling must be started before it can be stopped. The endpoint index can be obtained from the "device_state" command. This command returns an error if the "start_polling" has not been executed on the endpoint.
set_policy	Sets the pipe policy. (Each pipe has a set of pipe policies. The policy allows the system software to change the behavior of the pipe.) The height must be opened using "open_pipe" before the policy can be set. The two policy fields which can be set by the user are pp_max_outstanding_request and pp_periodic_max_transfer_size.
get_policy	Used to read the pipe policy. The pipe must be opened using the "open_pipe" command before the policy can be read.
reset	Used to clear and released the associated resources allocated by the software. The pipe must be opened using the "open_pipe" command before the pipe can be reset.
clear_pipe	Used to perform a pipe_reset on the control endpoint. This command is usually used when there is a stall condition.
set_private	Used to set a private data area in the USB client driver. This command is used to verify the usb_pipe_set_private USB architecture interface function.
get_private	Used to obtain private data that was set in the USB client driver. The set_private command must be used before the get_private command. The get_private command returns an error if the pipe is not been opened or if the set_private command has not been executed.
reserve	Used to reserve a pipe of an endpoint index by calling usb_pipe_reserve in the USBI module of the USB client driver. The pipe of the endpoint must be opened before the pipe can be reserved.
release	Used to release a pipe of an endpoint index. This command calls usb_pipe_release. The pipe of an endpoint must be opened and reserved before the pipe can be released.
get_addr	Used to perform the usb_get_addr USB architecture interface call on an endpoint index. The pipe must be opened before this command can be used.
interface#	Used to perform the usb_get_interface_number USB architecture interface call on an endpoint index. The pipe must be opened before this command can be used.
get_dev_desc	Used to perform the usb_get_dev_desc USB architecture interface call on an endpoint index. The pipe must be opened before this command can be used.

TABLE 3-continued

raw_config	Used to perform the usb_get_raw_config USB architecture interface call on an endpoint index. The pipe must be opened before this command can be used.

In addition to the foregoing commands, the test application provides for the following miscellaneous commands (see Table 4.)

TABLE 4

help/?	Invokes the help utility of the test application.
quit/exit	Terminates the test application.
sizeof	Returns data structure size for device, configuration or endpoint descriptors.

While the present invention has been described with reference to particular embodiments, it will be understood that the embodiments are illustrated and that the invention scope is not so limited. Any variations, modifications, additions and improvements to the embodiments described are possible. These variations, modifications, additions and improvements may fall within the scope of the invention as detailed within the following claims.

What is claimed is:

1. A test system for determining compliance of a universal serial bus (USB) system to a set of predetermined specifications comprising:

a computer system wherein said computer system is configured to execute a test application and USB system software and wherein said computer system includes a host controller;

a USB interconnect coupled to said computer system and configured to be controlled by said host controller;

wherein said computer system is configured to accept test commands from a user, execution of each of said test commands being associated with a corresponding USB system operation, said operation being performed upon execution of said test command.

2. The test system of claim 1 wherein said computer system comprises a Unix operating environment, wherein said test application comprises a command line interpreter configured to accept said test commands via individually entered command lines and wherein said user may enter both Unix commands and said test commands, said test commands being executed by said test system and said Unix commands being executed by a Unix shell.

3. The test system of claim 1 wherein said test application is configured to perform a predetermined series of said operations in response to a user input.

4. The test system of claim 3 wherein said series of operations comprises a set of USB standard device requests.

5. The test system of claim 3 wherein said series of operations comprises a set of USB architecture functions.

6. The test system of claim 1 further comprising a communications link between said computer system and a remote location and wherein said test commands are entered into said computer system from said remote location via said communications link.

7. The test system of claim 6 wherein data generated as a result of execution of said test command is transmitted to said remote location via said communications link.

8. The test system of claim 1 further comprising one or more USB devices coupled to said USB interconnect.

9. The test system of claim 8 wherein said test application is configured to enumerate said one or more USB devices and to display data representative of said enumerated USB devices to said user, and wherein said test application is further configured to enable selection of one of said enumerated USB devices.

10. The test system of claim 8 wherein said test application is configured to enumerate said one or more USB devices and wherein said test commands are configured to designate a particular one of said USB devices for said operation.

11. The test system of claim 10 wherein each of said USB devices has one or more endpoints and wherein said test commands are configured to designate a particular one of said endpoints for said operation.

12. The test system of claim 8 wherein said test application and said USB system software are configured to recognize attachment of one or more additional USB devices to said USB interconnect and wherein said test application and said USB system software are configured to recognize removal of one or more of said USB devices to said USB interconnect.

13. The test system of claim 8 wherein said test application is configured to store state information associated with at least one of said USB devices.

14. The test system of claim 8 wherein said test application is configured to establish one or more pipes between said test application and one or more corresponding endpoints in said USB devices.

15. The test system of claim 1 wherein said USB system operation is a multi-step function and wherein said test system is configured to execute said USB system operation in a single-step mode.

16. The test system of claim 15 further comprising a logic analyzer coupled to said USB interconnect, said logic analyzer being configured to generate data indicative of signal traffic on said USB during execution of said USB system operation.

17. A method for testing a function of a universal serial bus (USB) system of a computer, said USB system having a USB interconnect, one or more USB interfaces and one or more USB devices, the method comprising:

entering a test command on said computer;

interpreting said test command using a command line interpreter;

executing an operation associated with said test command;

transmitting a test signal to said USB system; and

validating a test result generated by said USB system in response to said test signal.

18. The method of claim 17 wherein said operation is a USB interface function and wherein said test signal is transmitted to one of said USB interfaces of said USB system.

19. The method of claim 17 wherein said operation is a standard device request and wherein said test signal is transmitted to one of said one or more USB devices of said USB system.

20. The method of claim 17 wherein said computer system is coupled to a remote location via a communications link and wherein entering said test command comprises entering said test command at said remote location and transmitting said test command over said communications link to said computer system.

21. The method of claim 17 further comprising entering a shell command on said computer and executing said shell command in an operating system shell running on said computer.

22. The method of claim 17 wherein executing said operation occurs in a single-step mode.

23. A method for testing a universal serial bus (USB) system, the USB system including a host computer system, a USB interconnect coupled to the host computer system and one or more USB devices coupled to the USB interconnect, the method comprising:

creating a node corresponding to one of said one or more USB devices;

opening said node;

obtaining state information for said one of said one or more USB devices;

storing said state information;

executing one or more tests based on said state information, each of said one or more tests corresponding to a particular function of said USB system; and

validating the results of said one or more tests.

24. The method f claim 23 wherein creating said node comprises enumerating said USB system to identify said one or more USB devices and creating nodes for said devices.

25. The method of claim 24 wherein obtaining state information for said one of said one or more USB devices comprises requesting one or more descriptors from said one of said one or more USB devices.

26. The method of claim 25 wherein said tests comprise tests of USBAI functions.

27. The method of claim 26 wherein said tests further comprise tests of standard device request functions.

28. The method of claim 25 wherein said tests correspond to one or more command lines entered by a user and interpreted by a command line interpreter.

29. The method of claim 28 wherein entering said command lines further comprises transmitting said command lines to said computer system from a remote location.

* * * * *

US006591310B1

(12) **United States Patent**
Johnson

(10) **Patent No.: US 6,591,310 B1**
(45) **Date of Patent: Jul. 8, 2003**

(54) **METHOD OF RESPONDING TO I/O REQUEST AND ASSOCIATED REPLY DESCRIPTOR**

(75) Inventor: **Stephen B. Johnson**, Colorado Springs, CO (US)

(73) Assignee: **LSI Logic Corporation**, Milpitas, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/569,715**

(22) Filed: **May 11, 2000**

(51) **Int. Cl.**[7] G06F 13/14; G06F 13/20

(52) **U.S. Cl.** 710/3; 710/5; 710/33; 710/48

(58) **Field of Search** 710/3, 5, 33, 48

(56) References Cited

U.S. PATENT DOCUMENTS

4,803,622	A *	2/1989	Bain et al.	710/5
5,640,599	A *	6/1997	Roskowski et al.	710/106
5,812,825	A *	9/1998	Ueda et al.	703/23
6,205,508	B1 *	3/2001	Bailey et al.	710/260
6,356,886	B1 *	3/2002	Doyle	706/46
6,430,596	B1 *	8/2002	Day, II	709/202

OTHER PUBLICATIONS

Intelligent I/O Architecture Specification, Version 2.0 (Feb. 11, 1999).

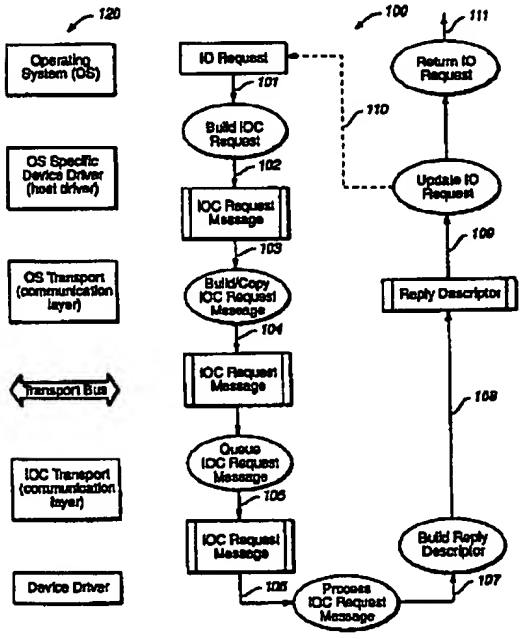
* cited by examiner

Primary Examiner—Jeffrey Gaffin
Assistant Examiner—Rehana Perveen
(74) *Attorney, Agent, or Firm*—Macheledt Bales, LLP

(57) **ABSTRACT**

A reply descriptor for transmission over an I/O message passing medium in response to a corresponding request message, the descriptor comprises at least one indication field that can function as a 'flag' to identify its type, and a content field; whereby a reply message is generated only if at least one predefined condition is not met and the content field will, accordingly, comprise information of that reply message's storage location. The content field to comprise data copied from the I/O request message if each predefined condition is met. A method of responding over an I/O message passing medium to a request message comprising the steps of: generating a reply message to the request message only if at least one predefined condition is not met; generating a reply descriptor having at least one indication field and a content field; whereby the content field comprises information of the reply message's storage location if so generated. Also, a program code on a computer readable storage medium comprising: a first program sub-code for generating a reply message to a corresponding I/O request message only if at least one predefined condition is not met. The first program sub-code comprising instructions for generating a reply descriptor having at least one indication field and a content field that comprises information of the reply message's storage location if said reply message is so generated.

25 Claims, 8 Drawing Sheets



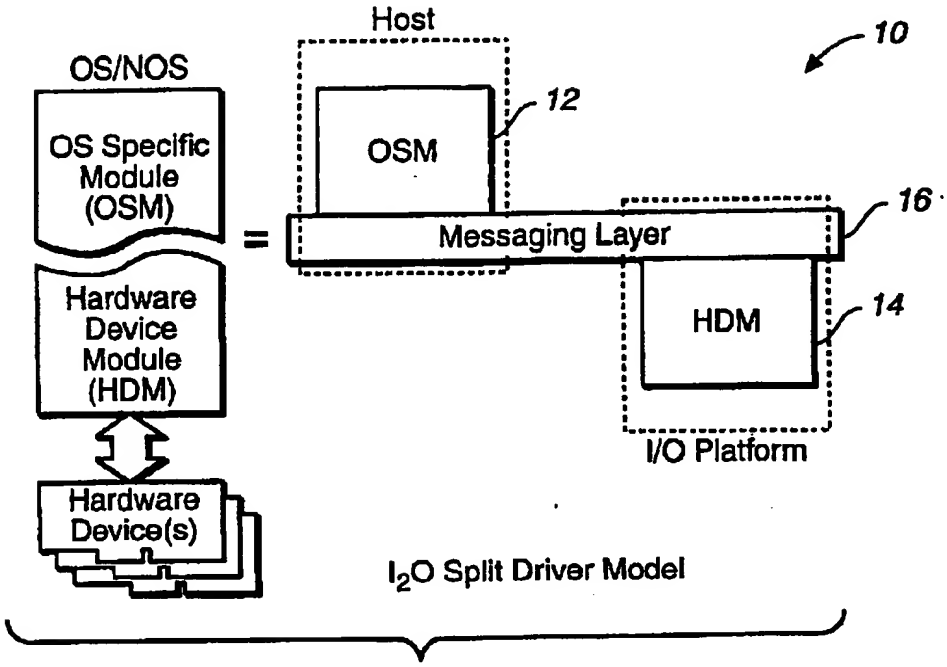

Host
10
OS/NOS
OS Specific Module (OSM)
OSM
12
Messaging Layer
16
Hardware Device Module (HDM)
HDM
14
I/O Platform
Hardware Device(s)
I2O Split Driver Model

FIG._1

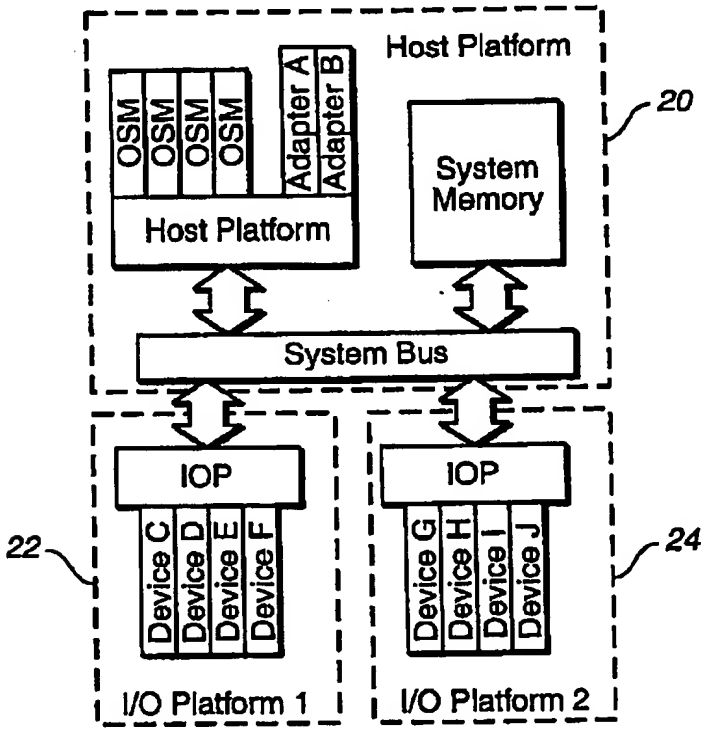

Host Platform
20
OSM
OSM
OSM
OSM
Adapter A
Adapter B
Host Platform
System Memory
System Bus
IOP
IOP
Device C
Device D
Device E
Device F
Device G
Device H
Device I
Device J
22
24
I/O Platform 1
I/O Platform 2

FIG._2A

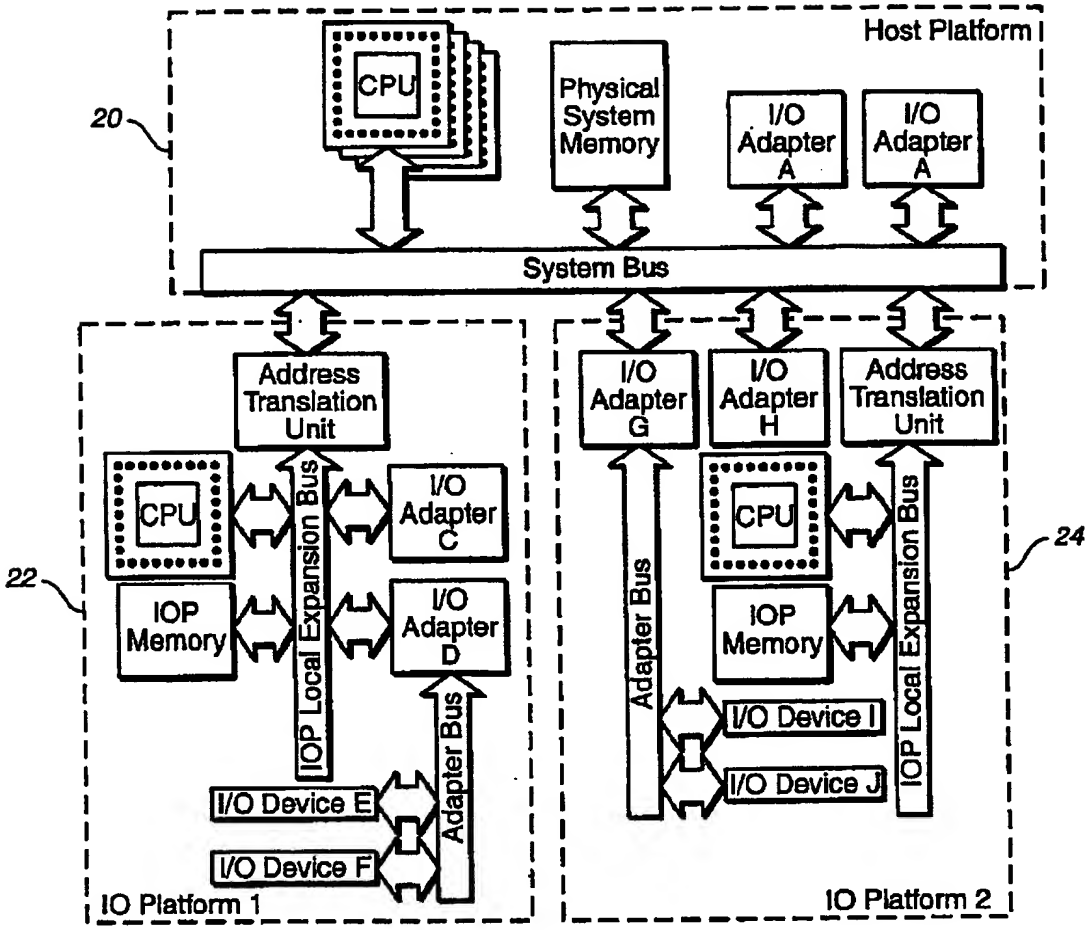

Host Platform
20
CPU
Physical System Memory
I/O Adapter A
I/O Adapter A
System Bus
Address Translation Unit
I/O Adapter G
I/O Adapter H
Address Translation Unit
CPU
I/O Adapter C
IOP Memory
I/O Adapter D
IOP Local Expansion Bus
Adapter Bus
22
24
I/O Device E
I/O Device F
I/O Device I
I/O Device J
IO Platform 1
IO Platform 2

FIG._2B

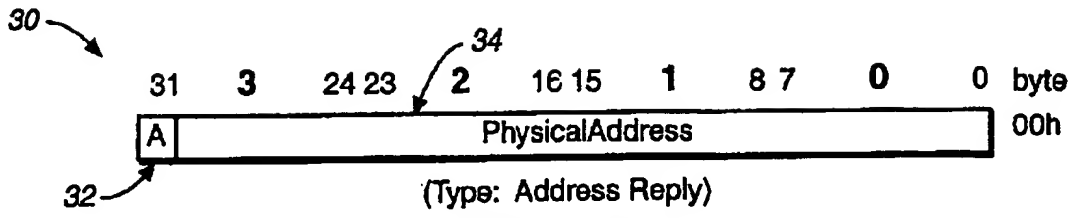


FIG._3A

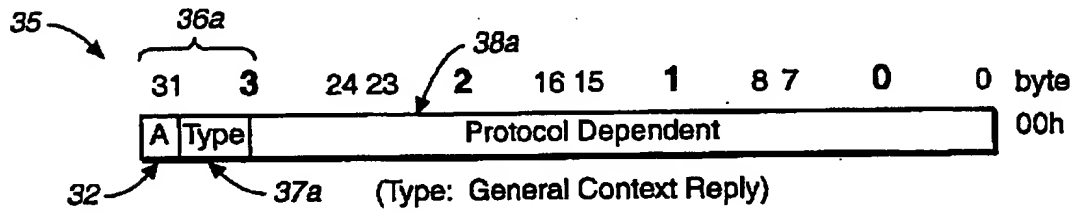


FIG._3B

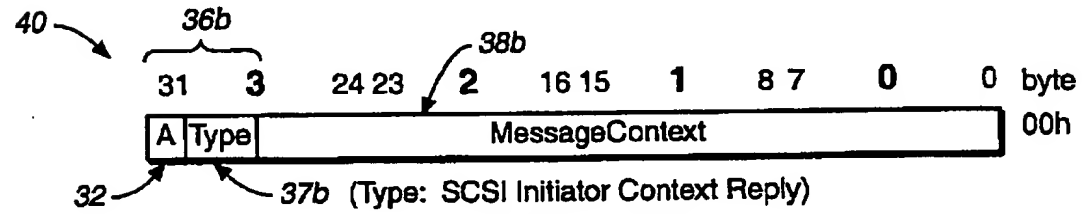


FIG._3C

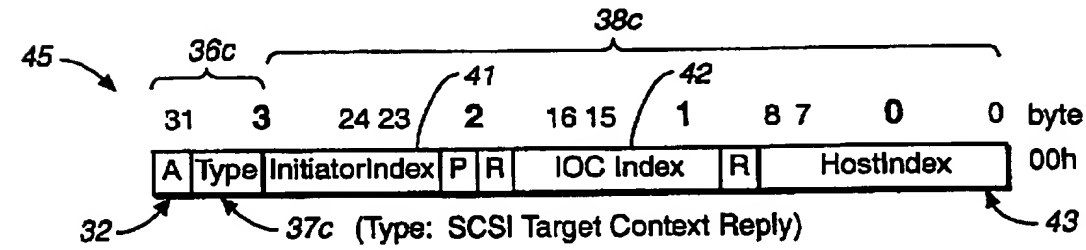
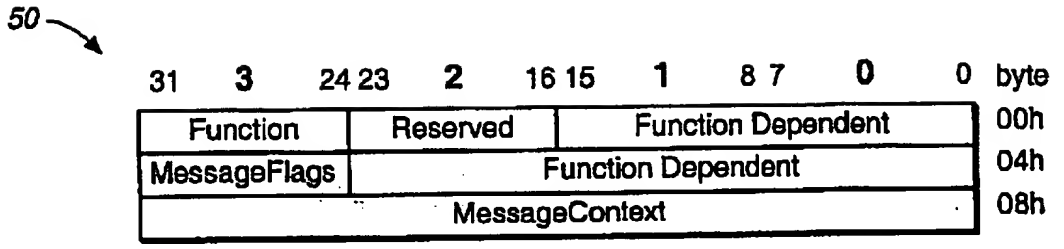
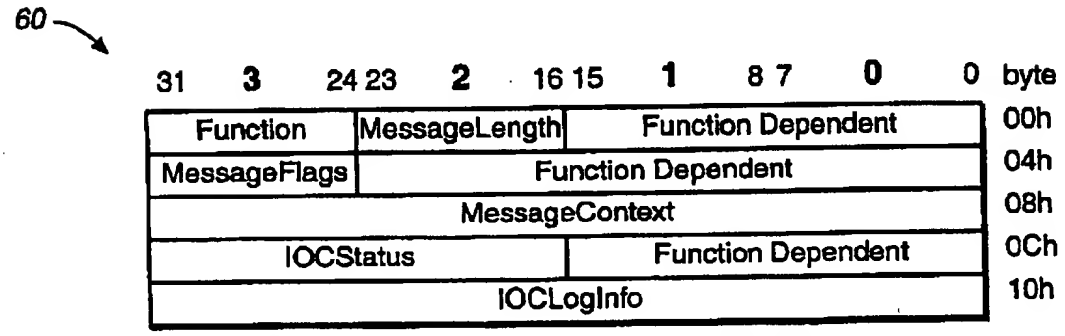


FIG._3D



Message Header (the first 12 bytes of every message frame)

FIG._3E



Default Reply Message (used when I/O status details are to be communicated)

FIG._3F

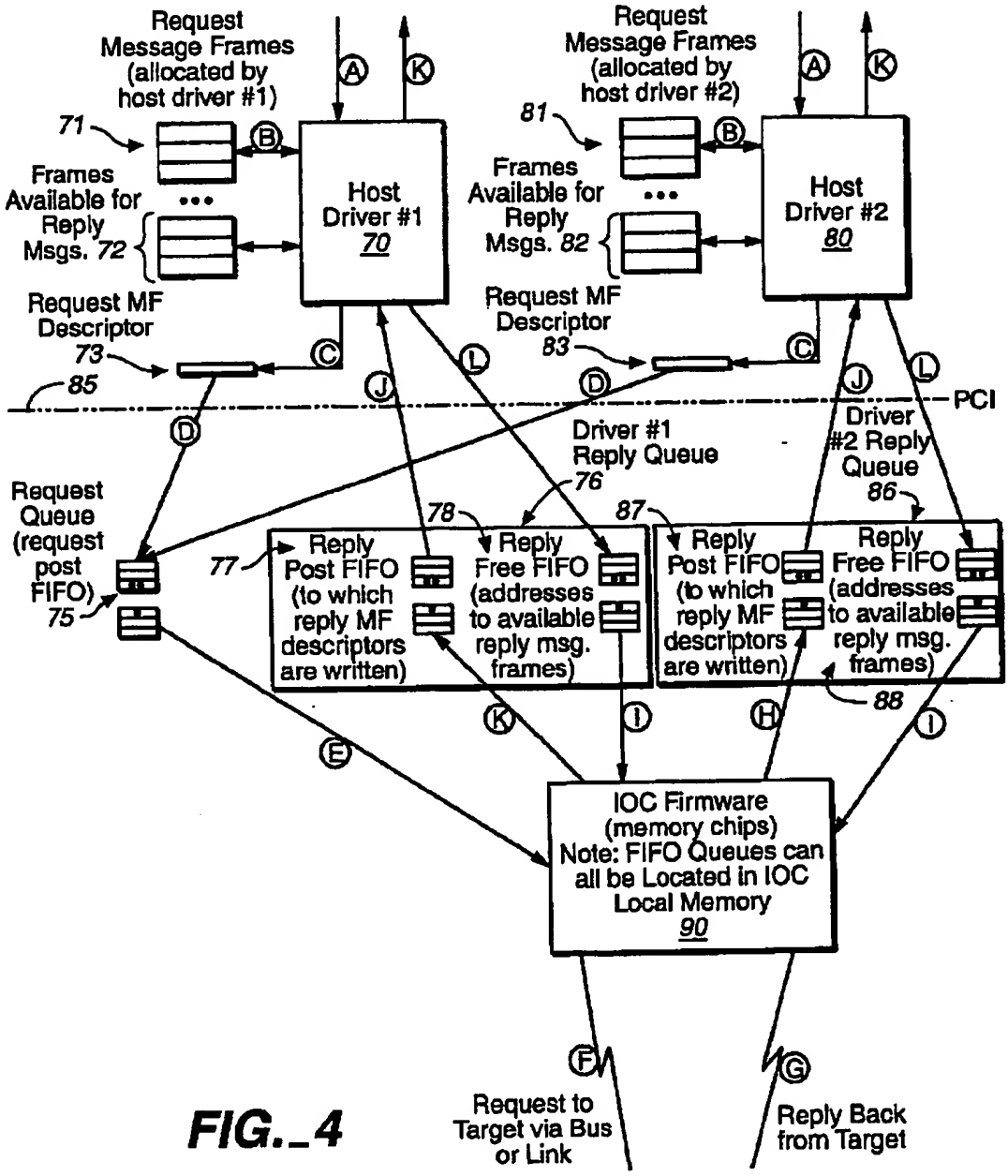

Request Message Frames (allocated by host driver #1)
71
Frames Available for Reply Msgs. 72
Request MF Descriptor
73
85
Host Driver #1 70
Request Message Frames (allocated by host driver #2)
81
Frames Available for Reply Msgs. 82
Request MF Descriptor
83
Host Driver #2 80
PCI
Driver #1 Reply Queue
76
Driver #2 Reply Queue
86
Request Queue (request post FIFO)
75
77
78
87
Reply Post FIFO (to which reply MF descriptors are written)
Reply Free FIFO (addresses to available reply msg. frames)
88
IOC Firmware (memory chips) Note: FIFO Queues can all be Located in IOC Local Memory 90
Request to Target via Bus or Link
Reply Back from Target

FIG._4

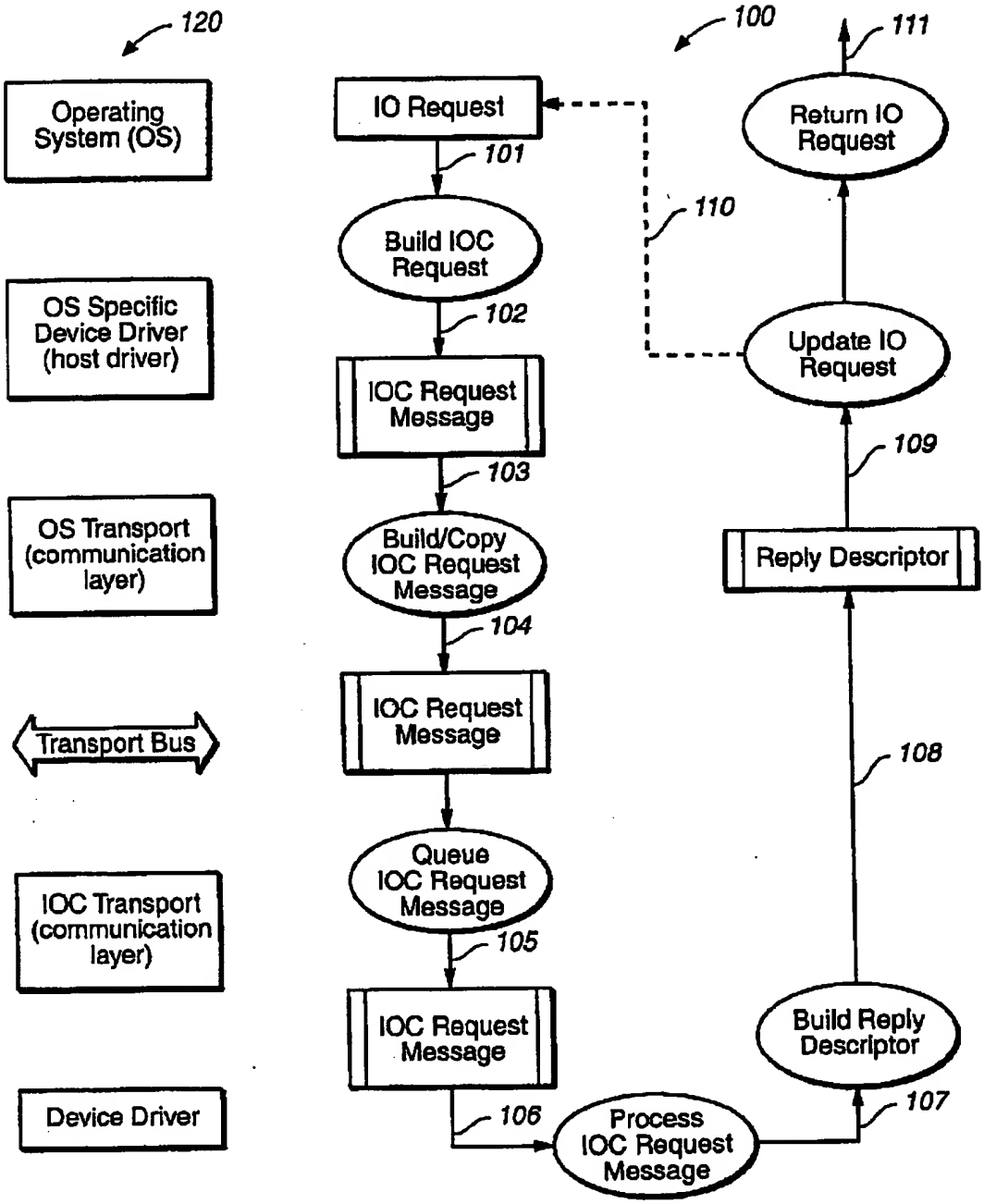
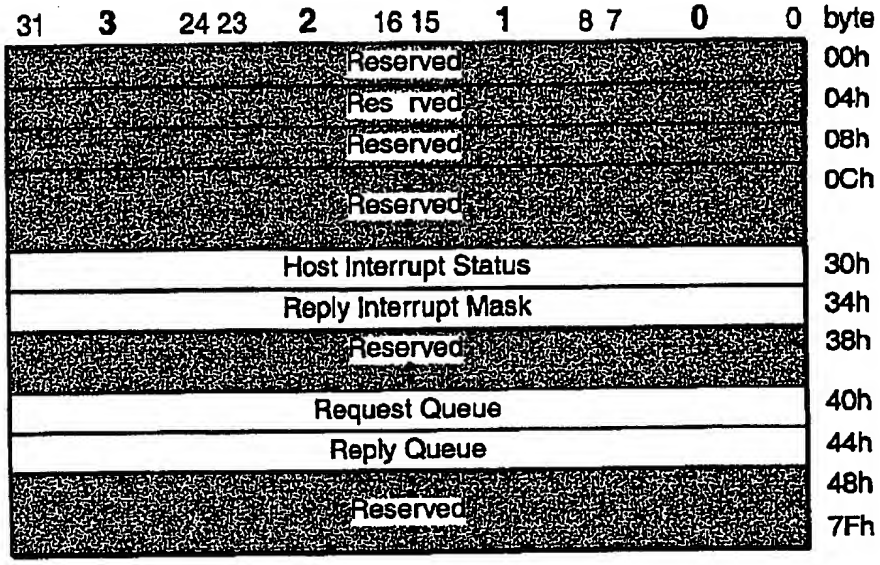

120
100
111
Operating System (OS)
IO Request
Return IO Request
101
110
Build IOC Request
OS Specific Device Driver (host driver)
102
Update IO Request
IOC Request Message
109
103
OS Transport (communication layer)
Build/Copy IOC Request Message
Reply Descriptor
104
IOC Request Message
Transport Bus
108
Queue IOC Request Message
IOC Transport (communication layer)
105
IOC Request Message
Build Reply Descriptor
Device Driver
106
107
Process IOC Request Message

FIG._5

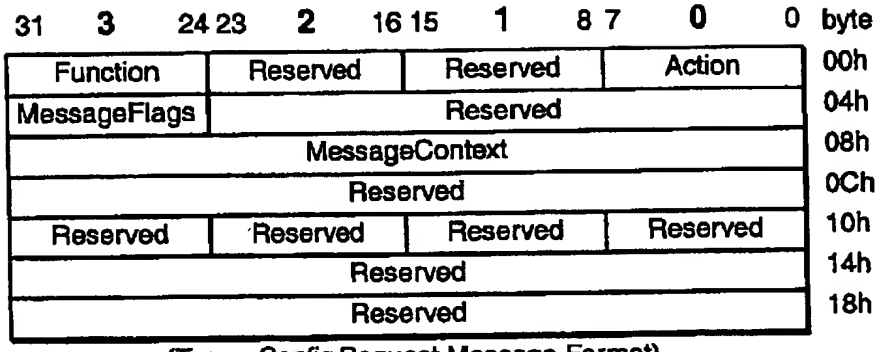
130



System Interface Register Msp

FIG._6

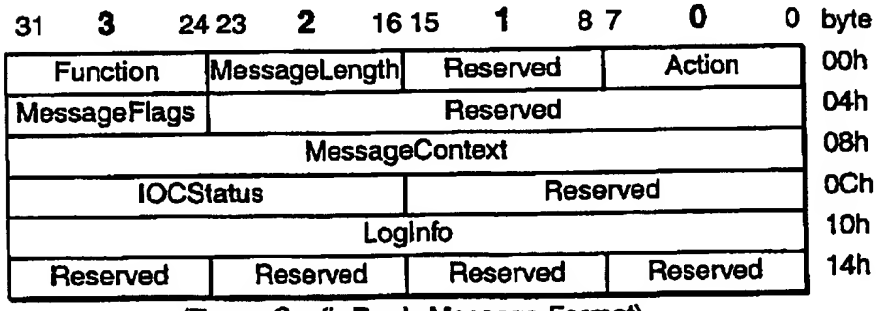
140



(Type: *Config* Request Message Format)

FIG._7A

150



(Type: *Config* Reply Message Format)

FIG._7B

160

31 3	24 23 2	16 15 1	8 7 0	0 byte
Function	Reserved	Bus	TargetID	00h
MessageFlags	Reserved	Reserved	CDBLength	04h
MessageContext				08h
LUN				0Ch
				10h
Reserved				14h
CDB				(msb) 18h
•••				1Ch
•••				20h
(lsb) (16 Bytes)				24h
Reserved				28h
Reserved				2Ch
SGL				30h

(Type: SCSI IO Request Message Format)

FIG._8A

170

31 3	24 23 2	16 15 1	8 7 0	0 byte
Function	MessageLength	Bus	TargetID	00h
MessageFlags	Reserved		CDBLength	04h
MessageContext				08h
IOCStatus		Reserved	Reserved	0Ch
LogInfo				10h
Reserved				14h
Reserved				18h
Reserved				1Ch

(Type: SCSI IO Error Reply Message Format)

FIG._8B

METHOD OF RESPONDING TO I/O REQUEST AND ASSOCIATED REPLY DESCRIPTOR

BACKGROUND OF THE INVENTION

In general, the present invention relates to communicating message information, or data, over message passing interface(s) between program modules, such as a target peripheral device module and an operating system (OS) module within an I/O system, whether the modules are executed on the same or different digital computer processors and whether utilizing different operating systems. Of particular interest is the message reply portion of the communication between I/O system modules to, eventually, send status information back to the caller (e.g., an operating system) that issued the original command, regardless of the specific communications protocol and interface technology employed. More particularly, the invention relates to a unique method of responding over an I/O message passing medium to a corresponding request message, by way of an associated novel reply descriptor transmitted in response thereto.

Within an I/O system, typical computer hardware includes a host system entity connected to communicate with one or more I/O devices. The trend in development of I/O system architecture is to utilize a split driver model, see FIG. 1, as explained more-fully in a Version 2.0 of the "Intelligent I/O Architecture Specification" dated Feb. 11, 1999 (herein referred to as simply "I_2O Specification"), the written work product of the collaborative effort of several commercial entities including the applicant hereof. Within the split driver model two basic software modules are defined, each of which can execute on different physical processors and within different operating environments: (1) an OS-specific module (OSM) which provides an interface to the operating system (OS); and (2) a hardware device module (HDM) which provides an interface to each I/O adapter and corresponding device. These two basic modules intercommunicate via a logical "messaging layer" comprised of a network of MessengerInstances (depicted in FIGS. 2–3 of the I_2O Specification) as illustrated herein in FIG. 1 over which request messages to I/O devices and completion reply messages are transmitted to effect commands from the operating system rather than having the host/OSM directly read and write from and to each I/O device register. The split driver model allows for expansion of the I/O system through software development, independent of both device hardware and the operating system.

I. Conventional Way to Respond to Requests per I_2O Specification

Additional layers of stackable drivers beyond the basic OSM and HDM can be logically defined as has been done in the I_2O Specification to provide additional functionality between the two basic program modules. The stacking of drivers increases the request and reply message load of the system, in turn decreasing its speed/performance. For example when operating within I_2O, on the order of 28,000 I/O messages are transmitted per second. The I_2O Specification explains in Section 3.4.1 ("Message Structure and Definitions"), messages are data structures that contain a fixed-size header containing device address and payload description and, immediately following, a variable-size payload containing all additional information associated with the message. If the payload refers to memory, a scatter-gather list (SGL) is included in a format understandable by the originator, the target, the transport, and any intermediate

software layers. The header and payload parts reside within a physically-contiguous buffer called the message frame buffer (such as is shown in FIGS. 3–19 of the I_2O Specification).

I_2O messages fall into two basic categories: (1) request messages initiate activity at the destination (a request may contain multiple transactions of the same type); and (2) reply messages return status information concerning one or more requests. According to I_2O convention, a reply message is generated and sent for every request (see I_2O Specification sections 6.1.2 and 6.4.4), regardless of whether the request was completed without error (see section 6.4.4.2.1 of I_2O).

I_2O convention classifies all messages, each class has a format for request messages and a protocol for generating and transmitting reply messages for that class. For example, 'utility messages' are common to all message classes, and messages specific to a particular message class are 'base class messages'. According to the I_2O Specification, inbound and outbound queues are reserved for each I/O platform (referred to therein as "IOP"—see FIGS. 2A and 2B schematically outlining the relationship between a host platform, an IOP1, and IOP2). Note that the I_2O Specification uses IOP synonymously with an 'I/O processor entity' dedicated to processing I/O transactions (consisting of processor, memory, and I/O devices). The inbound queue of an IOP receives messages from all other platforms, including the host system, and the outbound queue of each IOP collectively function as an input queue for the host system. Thus, each IOP provides support for passing messages without requiring additional host system hardware. Once IOPs establish connection, the program modules at each end of the connection can send and receive messages (generally in an asynchronous fashion as non-blocking by nature). In the specific case of an SCSI Controller, for example, it is the SCSI hardware device module that detects and registers devices connected to the SCSI bus—and these devices are accessible through messages passed through the SCSI hardware device module.

According to I_2O section 6.1.2, reply messages fall into two general categories (as identified by the REPLY bit in the message header's MessageFlags field): "failed" messages and "processed" messages. Failed messages are those that cannot be processed (including messages that cannot be delivered or contain invalid or missing data). A request message "fails" when the message layer cannot deliver the message or the target device does not understand the format of the request (e.g., unknown message version). Section 6.1.2.1 further distinguishes a failed message from one that is processed but is unable to be successfully completed due to "error": The inability of a device driver module (DDM) to perform or carry out the request is referred to as an "error". Thus, a successfully completed request is one that is processed without error. Note that in I_2O, the acronym DDM is often used generically in place of specifying whether the module is a hardware device module (HDM) or an intermediate service module (ISM).

Section 3.4.1.2.2 of I_2O specifies the template for a "normal single transaction reply message" as shown in FIGS. 3–23; and section 3.4.3.2 identifies a "multiple transaction reply message" model (wherein one or more successful transactions may be combined into one reply message, see section 6.4.4.2.2 of I_2O).

As shown and explained in more detail in Chapter 2 of the I_2O Specification (pages 2–19 through 2–22), whether request messages are sent from the host system to a hardware device module or are sent peer-to-peer (from one hardware device module to another), all reply messages built

include the Initiator Address, Target Address, and the Initiator Context field from the request message. Once the reply message is built, the hardware device module calls a respective message service. The sending entity allocates a reply message frame, copies the reply message into the frame and places/writes the frame's address in the appropriate message queue. I_2O FIGS. 2–13 diagrams the flow of events for its conventional process of sending request and reply messages (I_2O Specification explanation reproduced below, for reference):

1. The operating system issues an I/O request.
2. The OSM (Operating System Module) accepts the request and translates it into a message addressed to the DDM (I_2O uses the acronym DDM generically for hardware device module, HDM, or intermediate service module, ISM). The Initiator Context field is set to indicate the message handler for the reply. The OSM has the option to place a pointer to the OS I/O request in the message's transaction context field.
3. The OSM invokes the communication layer to deliver the message.
4. The host's MessengerInstance (a collection of services that support initializing, configuring, and operating its client modules, see FIGS. 2–3 of the I_2O Specification) queues the message by copying it into a message frame buffer residing on the remote IOP.
5. The IOP on the other end posts the message to the DDM's event queue.
6. The DDM processes the request.
7. After processing the message and satisfying the request, the DDM builds a reply, copies the initiator's context and transaction context fields from the request to the reply, addresses the reply to the initiator, and finally invokes the message service to send it to the originator of the request.
8. The IOP's message service queues the reply by copying it into a message frame buffer residing at the host's MessengerInstance.
9. The IOP alerts the host's MessengerInstance to the message ready for delivery.
10. The host's MessengerInstance invokes the OSM's message handler with the reply.
11. The OSM retrieves the pointer to the OS I/O request from the message's transaction context field to establish the original request context and completes the OS I/O request.
12. The driver returns the request to the OS.

II. For Reference: Brief Background of SCSI

The widely-used small computer system interface (SCSI) protocol was developed for industry groups, under the American National Standards Institute (ANSI) and International Standards Organization (ISO) guidelines, to provide an efficient peer-to-peer I/O bus. Devices that conform with the mechanical, electrical, timing, and protocol requirements (including the physical attributes of I/O buses used to interconnect computers and peripheral devices) of the SCSI parallel interface will interoperate. This allows several different peripherals (hard disk drives, removable disk drives, tape drives, CD-ROM drives, printers, scanners, optical media drives, and so on) to be added at the same time to a host computer without requiring modifications to the generic system hardware. The working draft of the SCSI Parallel Interface-2 Standard (SPI-2), as modified (Rev. 16, dated Oct. 14, 1997), defines the cables, connectors, signals, transceivers, and protocol used to interconnect SCSI devices. The SPI-2 working draft states that a SCSI bus consists of all the conductors and connectors required to attain signal line continuity between every driver, receiver, and terminator for each signal. In operation, a SCSI bus is a bidirectional, multimaster bus which can accommodate peer to peer communications among multiple computer processing units (CPUs) and multiple peripheral devices. A SCSI device is one that contains at least one SCSI port and the means to connect the drivers and receivers to the bus.

A SCSI primary bus is one that provides for and carries 8-bit or 16-bit data transfer. A SCSI secondary bus carries an additional 16-bit data bus that, when used in conjunction with a 16-bit primary bus, provides for a 32-bit data transfer path (although the latter is not, yet, widely used). SCSI devices may connect to a bus via 8-bit, 16-bit, or 32-bit ports. To date, SCSI parallel interface devices may be implemented with 50, 68, or 80 pin connectors (whether shielded or unshielded). As is known, a typical data transfer operation over a SCSI bus between a SCSI controller (or "host adapter") located in a host computer system, to a target device (such as a disk drive) has seven SCSI "phases": (1) ARBITRATION, (2) SELECTION, (3) RESELECTION, (4) COMMAND, (5) DATA, (6) STATUS and (7) MESSAGE. For example, during the COMMAND phase, a SCSI command is transferred from the host adapter to a target (drive), and so on. Host adapter functional circuitry is typically maintained on a host bus adapter (HBA) chip on a printed circuit board structure referred to as a host adapter board (HAB) for connection to a PC host via an expansion slot.

III. For Reference: Brief Background of Fibre Channel Interconnect

Fibre Channel is a newer interface technology (emerging along with Serial Storage Architecture (SSA) and IEEE P1394) capable of transferring data as fast and faster than an Ultra3 SCSI system can, over fiber optic cabling as well as copper transmission media. Fiber Channel-type host bus adapters are installed into a host computer expansion slot just as SCSI host bus adapters are installed. Fibre channel connections are often associated with the term "loop" (from the name Fibre Channel arbitrated loop) rather than "bus" (as SCSI devices are connected). There are actually other types of Fibre Channel connections, called "point to point" and "fabric." With fibre channel, communication between hosts and devices does not have to be done directly. Instead, users can employ hubs and switches between devices on the Fibre Channel network. Hubs and switches can be used to create Fibre Channel "storage networks". Fibre Channel cabling can be copper (can be up to 30 meters in length) or fiber optic (currently up to 10 Km). In addition, no termination is necessary for fibre channel as is required in SCSI. Fiber optic ports directly placed on a peripheral device allow only for connections to fiber optic cabling. Commercially available adapters exist that allow a SCSI-compliant device to be connected to a Fiber Channel loop.

IV. Identification of New Method and Reply Descriptor

The trend in I/O architecture development is toward stacking more layers of logical drivers and creating high performance systems, thus increasing the request and reply message overhead of the I/O system, which in turn decreases system efficiency and performance. Therefore, a new useful method of responding to an I/O request message and associated reply descriptor are needed to make messaging between one or more hosts, one or more interconnected devices, or any host and an interconnected device, more efficient. Without a reasonable, reliable, and cost-effective solution at hand for increasing I/O system performance,

computer hardware and software developers will find it very difficult to meet the demand for managing more devices in ever-complex I/O environments. As anyone who depends on computerized systems to accurately and efficiently perform selected tasks will readily understand: It is imperative that valuable I/O messaging data be communicated in a reliable, efficient manner that reduces system I/O overhead by using less system resources such as system memory, CPU (computer processing unit) cycles, system bus resources, and so on.

SUMMARY OF THE INVENTION

It is a primary object of this invention to provide a method of responding over an I/O message passing medium, to a request message and to provide an associated reply descriptor for transmission over an I/O message passing medium in response to a corresponding request message. A reply message need only be generated if at least one predefined condition is not met, the reply descriptor to include at least one indication field that identifies its type and a content field, whereby the content field comprises information of the reply message's storage location (in the event so generated). It is a further object to provide computer executable program code on a computer readable storage medium, having instructions for carrying out the novel method and generating the novel reply descriptor.

The simple, efficient design of the invention allows the innovative method, reply descriptor, and program code as contemplated hereby, to be tailored-to, readily installed, and run using currently-available processors, memory, communications protocol and interface types, as well as those under, or contemplated for, development. Further, unlike the conventions currently specified and in use, the unique method, reply descriptor, and program code of the invention do not require that full reply message(s) be built, copied, read and processed for each and every request message processed. In the spirit of this unique design, a reply descriptor can be generated and then transmitted for a corresponding request for any class of messages as will be further appreciated.

Although the advantages of providing the flexible new method, associated new reply descriptor, and program code, as described herein, will be more-fully appreciated in connection with the full specification, certain advantages are listed as follows:

(a) System Cost Reduction and Process Simplification—For the vast majority of request/commands generated by an initiator which are successfully completed (processed without error), a full reply message need not be build, copied, and read as is conventionally done; thus, reducing adapter CPU firmware cycles necessary to manage queues, which in turn, requires less expensive adapters to handle the reply overhead. Unlike conventional protocol, this powerful novel method and associated reply descriptor allows one to simplify the process to communicate I/O request completion. Simplifying the design reduces overall system costs, such as the cost. Reducing system costs, in-turn reduces the cost to perform important I/O messaging functions.

(b) Design Flexibility and Versatility—The invention can accommodate many different message passing medium hardware interface types; a wide variety of communications protocols and message templates; and a multitude of different types of I/O systems and devices. Furthermore, many different computer platforms can readily use the more-flexible I/O reply solutions offered by the instant invention.

Briefly described, again, the invention includes a reply descriptor for transmission over an I/O message passing medium in response to a corresponding request message. The reply descriptor comprises at least one indication field that can function as a 'flag' to identify type of the reply descriptor, and a content field; whereby a reply message is generated only if at least one predefined condition is not met and the content field will, accordingly, comprise information of that reply message's storage location. Also characterized is a method of responding over an I/O message passing medium, to a request message. The method comprises the steps of: generating a reply message to the request message only if at least one predefined condition is not met; generating a reply descriptor having at least one indication field and a content field; whereby the content field comprises information of the reply message's storage location if it is so generated.

Further characterized is a computer executable program code on a computer readable storage medium. The program code comprises: a first program sub-code for generating a reply message to a corresponding I/O request message only if at least one predefined condition is not met. The first program sub-code comprising instructions for generating a reply descriptor having at least one indication field and a content field that comprises information of the reply message's storage location if said reply message is so generated. The content field to comprise data copied from the I/O request message if each predefined condition is met.

Additional, further distinguishing associated features of the reply descriptor, method, and program code of the invention will be readily appreciated as set forth herein, including the following novel features. The message passing medium over which reply descriptors may be transmitted may comprise one or more parallel, serial, and wireless bus, or any hybrid thereof. More-specifically, suitable buses include those operational with any of a variety of hardware interface types such as SCSI (Small Computer System Interface), Fibre Channel, PCI (Peripheral Component Interconnect), PCI-X, ISA (Industry Standard Architecture), InfiniBand, IDE (Integrated Drive Electronics), USB (Universal Serial Bus), RS-232, EISA (Extended ISA), Local Bus, Micro Channel, and so on. Further, the message passing medium can utilize any number of communications protocols such as those identified as SCSI, ATM (Asynchronous Transfer Mode), IPI (Intelligent Peripheral Interface), HiPPI (HIgh Performance Parallel Interface), IP (Internet Protocol), InfiniBand, SSA (Serial Storage Architecture), IEEE P1394, and so on. Upon the writing of the reply descriptor to a reply-post buffer, an interrupt (or any suitable alert mechanism by which to signal a host that a reply descriptor has been so written) can be transmitted for a host-based driver to read the reply descriptor; and once so read, the host-based driver can correlate the reply descriptor with the request message and send a notification message on to an originating-caller (such as a host-based operating system).

If each such predefined condition is met, the indication field might further comprise a type field, and the content field can comprise data copied from and unique to the request message as generated by a host-based driver; but if the reply message is so generated, it preferably comprises data regarding the predefined condition(s) not met. The content field might further have a receiving port identifier. Also, especially in the event each such predefined condition is met, the data unique to the request message can comprise any f a number of identifiers such as: an address (whether physical or virtual) to a storage space in a memory (which

could be a temporary-storage, e.g. a queue, or more-permanent storage, e.g. a message frame), an index value or offset to a table, an index value or offset to a list, an index value or offset to a register, an index value r offset to a layer of hardware registers (stack), an index value or offset to an array, content-data associated with a hardware assisted CAM, and so on. To conserve computer resources, the alert signal may be transmitted to the host-based driver after a predetermined number of such reply descriptors have been generated.

In the event the reply message is so generated, the content field of the reply descriptor can comprise an address to an available reply frame buffer located in a host memory; this address having been removed from a plurality of addresses residing on a reply-free buffer, each such address to identify a location of a corresponding reply frame buffer. Once the reply descriptor has been read by a host-based driver, the host-based driver can be instructed to remove the reply descriptor from the reply-post buffer.

BRIEF DESCRIPTION OF THE DRAWINGS

For purposes of illustrating the flexibility of design and versatility of the innovative preferred method, associated reply descriptor, and program code, the invention will be more particularly described by referencing the accompanying drawings of embodiments of the invention (in which like numerals designate like parts). The figures have been included to communicate the features of the invention by way of example, only, and are in no way intended to unduly limit the disclosure hereof.

FIG. 1 is a schematic depicting a conventional split driver model 10 as explained in the "Intelligent I/O Architecture Specification" Version 2.0 (I_2O Specification). An OS-specific module (identified as "OSM" at 12) that provides an interface to an operating system ("OS") and a hardware device module (identified as "HDM" at 14) that provides an interface to each I/O adapter and corresponding device, which intercommunicate via a logical "messaging layer" 16.

FIGS. 2A and 2B (reproduced from the I_2O Specification for easy reference) schematically outline the relationship between conventional components of an I_2O segment (a host platform 20, an IOP1 at 22, and IOP2 at 24).

FIG. 3A is a schematic of the fields in a preferred general address reply descriptor of the invention.

FIG. 3B is a schematic of the fields in a preferred general context reply descriptor of the invention.

FIGS. 3C and 3D are schematics of the fields in two special cases of a preferred context reply descriptor of the invention.

FIG. 3E depicts the fields of an example "header" for a message.

FIG. 3F depicts an example default reply message such as can be used to communicate certain selected details of any one or more predefined condition not met.

FIGS. 4 and 5 are schematics detailing selected message and data flow features of an I/O system designed to aid in carrying out a preferred method of the invention, including flow of a preferred reply descriptor of the invention.

FIG. 6 depicts an example map of System Interface Registers through which a host can communicate with an IOC (I/O Controller)—preferably access to these registers is provided via memory and/or IO mapping.

FIGS. 7A and 7B depict alternative message formats, specifically detailing fields of an example Config Request Message and associated Config Reply Message.

FIGS. 8A and 8B depict alternative request and reply message formats for, respectively, a SCSI I/O Request Message and a SCSI I/O Error Reply Message.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

Two basic types of preferred reply descriptors of the invention are depicted in FIGS. 3A and 3B. Turn, first, to the Address Reply Descriptor 30 shown in FIG. 3A: It includes information 34 useful for identifying a storage location of the frame for a reply message generated in connection with an unsuccessful I/O and an address bit 32.

Description of Fields Represented in the Address Reply Descriptor of FIG. 3A

PhysicalAddress	Bits 1::31 of the physical address of the reply message frame generated, for example by the I/O Controller. This is a SMFA (System Message Frame Address) that has been shifted right by one bit making room for the address bit 32.
A	The address bit 32 is set to 1 to denote it is an Address Reply Descriptor (since at least one command/condition was not met, therefore unsuccessful I/O).

FIG. 3B depicts another general type of reply descriptor, namely, a context reply descriptor 35 that includes data preferably comprising at least some portion or sub-portion of data copied from the corresponding request message to which descriptor 35 pertains. For reference, the field containing this data copied from the request message is labeled "Protocol Dependent" 38*a*. As will be better appreciated in connection with FIGS. 4 and 5, the context reply descriptor mechanism is unique, especially in that no reply message need accompany it in responding to an I/O request. The fields labeled 32 and 37*a* collectively make up an indication field 36*a* encoded for an initial 'quick' identification of whether or not the descriptor is an address reply descriptor containing a pointer, or address, to a reply message (address bit set to 1); and if the descriptor is not of an address type, the specific type of context reply descriptor it is (for example, see Type Table below).

Description of Fields Represented in the General Context Reply Descriptor of FIG. 3B

Type Bits:		The specific type of Context reply message. Definition of Type bits (Type Table):
Bit 1	Bit 0	Type
0	0	SCSI Initiator mode Context Reply
0	1	SCSI Target mode Context Reply
1	0	Reserved
1	1	Reserved
A		The Address bit--when using a Context reply msg. this bit is reset to 0 (since there was successful completion of the I/O request).

When the IOC is a SCSI initiator, a context reply descriptor such as that in FIG. 3C may be used for replying to an I/O request message that has been processed without error. The receipt of the reply descriptor 40 as depicted, namely a SCSI Context Reply, by the host driver implies the successful completion of the I/O. In this example, the IOC has copied (digitally, or otherwise, duplicated) the MessageContext data directly from a corresponding request message to generate a content field (38*b*) for the reply descriptor 40

which is put on a Reply Post buffer (such as either of the FIFO's shown in FIG. 4 at 77 and 87). The Type (37*b*) and A (32) sub-fields of indication field 36*b*, are set once the MessageContext of the request message has been copied. In this example as shown, bits 0::28 of the MessageContext can be used by the host driver in any particular way selected.

Description of Fields Represented in the SCSI Initiator Context Reply Descriptor of FIG. 3C

MessageContext	A copy of the MessageContext field, bits 0:28, copied from that supplied by the host in the request message.
Type	The specific type of Context Reply-Set to denote a SCSI initiator mode Context Reply.
A	The Address bit, reset to 0 (since there was successful completion of the I/O request as in the case for general Context Reply 35).

The alternative reply descriptor 45 illustrated in FIG. 3D has more detailed information concerning origination and transmission of the corresponding request. The specific information selected for encoding in a context reply descriptor of the invention will depend upon, among other things, I/O architecture design factors. When the IOC is in "TARGET" mode and a SCSI command has been transmitted and received, the depicted SCSI Target Context Reply Descriptor 45 may be used to convey more specific information to the host system. Here, indication field 36*c* is comprised of fields labeled Type (37*c*) and A (32) and the protocol dependent content field 38*c* includes valuable additional information as follows:

Description of Fields Represented in the SCSI Target Context Reply Descriptor of FIG. 3D

HostIndex (43)	This index is specified by the host to track the I/O.
IOCIndex (42)	The index used by the IOC to track this I/O. This bit can be used to indicate on which port the initial command was received.
InitiatorIndex (41)	A 6-bit value indicating which initiator sent this command. For parallel SCSI systems this can be the actual initiator ID value. For FCP systems, it's an index into a table of logged in initiators.

-continued

Type	Specifies type or mode of this Context reply message.
A	The Address bit, here it has been reset to 0 (since there was successful completion of the I/O request).
R	Reserved bit.

As mentioned above, two types of messages are used to convey information within the I/O method and system of the invention: (1) request messages are created by the system to "request" an action by an IOC; and (2) reply messages are used by an IOC to send status information back to the system. Each message includes a message header (of any predefined size) and a payload. By way of example only, the message header 50 represented in FIG. 3E is the first 12 bytes of each message frame and reserved and unused fields will have a value of 0 (zero). The header includes information to uniquely identify the message.

Description of Fields Represented in the Message Header of FIG. 3E

Function Dependent	The format of this field is dependent on the function being described.
Reserved	Function dependent.
Function	The function number of this message. This number determines the format of the rest of the message (and differentiates each request message from the others).
MessageFlags	All reserved bits must have a selected value, such as 0, unless otherwise identified in specific messages.
MessageContext	A value used to uniquely identify this message. Created by the host driver and not modified by the IOC. This value can be copied and 'returned' in the Context Reply, see FIG. 3B at 38b.

I/O reply messages are currently used in each instance a response to an I/O request is transmitted back to an OSM (operating system module). The table below describes a default reply message such as that labeled 60 in FIG. 3F.

Description of Fields Represented in the Default Reply Message of FIG. 3F

IOCStatus	Description
IOCSTATUS_SUCCESS	Command completed successfully from the IOC standpoint.
IOCSTATUS_INVALID_FUNCTION	Function not supported by the IOC.
IOCSTATUS_BUSY	Can not process the request at this time.
IOCSTATUS_INVALID_SGL	SGE not supported or understood.
IOCSTATUS_MSG_XFER_ERROR	System bus error detected during message transf.
IOCSTATUS_DATA_XFER_ERROR	System bus error detected during data transfer.
IOCSTATUS_INSUFFICIENT_RESOURCES	The IOC has insufficient resources to process the request at this time.
IOCSTATUS_INVALID_FIELD	A field in the message has an invalid value
IOCSTATUS_CONFIG_BAD_ACTION	The action is not supported
IOCSTATUS_CONFIG_BAD_TYPE	The configuration type is not supported
IOCSTATUS_CONFIG_BAD_PAGE	The configuration page is not supported
IOCSTATUS_CONFIG_BAD_DATA	Incorrect field setting within the configuration data
IOCSTATUS_CONFIG_NO_DEFAULTS	Can not set defaults for this page
IOCSTATUS_CONFIG_CANT_COMMIT	Non-volatile memory not available or error while writing persistent data to non-volatile memory
IOCSTATUS_SCSI_RECOVERED_ERROR	I/O operation completed successfully after retries
IOCSTATUS_SCSI_INVALID_BUS	Out of range Bus value in request message
IOCSTATUS_SCSI_INVALID_TARGETID	Out of range TargetID value in request message
IOCSTATUS_SCSI_DEVICE_NOT_THERE	Selection time-out or device does not exist

-continued

IOCStatus	Description
IOCSTATUS_SCSI_DATA_OVERRUN	SCSI device attempted to transfer more data than the amount specified by the byte count
IOCSTATUS_SCSI_DATA_UNDERRUN	SCSI device transferred less data than the amount specified by the byte count
IOCSTATUS_SCSI_IO_DATA_ERROR	I/O terminated because of unrecoverable bus parity or CRC error
IOCSTATUS_SCSI_PROTOCOL_ERROR	I/O terminated because of unrecoverable bus protocol error
IOCSTATUS_SCSI_TASK_TERMINATED	I/O terminated because of SCSI Task Management Request
IOCSTATUS_SCSI_BUS_RESET	I/O terminated because of a Bus Reset unrelated to a SCSI Task Management Request
IOCSTATUS_SCSI_TASK_MGMT_FAILED	SCSI Task Management function failed
IOCSTATUS_TARGET_PRIORITY_IO	An I/O operation has been received that requires priority handling
IOCSTATUS_TARGET_INVALID_PORT	A command was directed to a port that does not exist on this IOC.
IOCSTATUS_TARGET_INVALID_IOCINDEX	The Target Host used an IocIndex value that is invalid or not in use on the IOC.
IOCSTATUS_TARGET_ABORTED	This is a reply for an I/O or buffer that was aborted at the request of the host.
IOCSTATUS_TARGET_NO_CONNECTION_RETRYABLE	Unable to communicate to the Initiator of this I/O. The host can retry this operation later.
IOCSTATUS_TARGET_NO_CONNECTION	Unable to communicate to the Initiator of this I/O. This I/O can never be continued.
IOCSTATUS_TARGET_FC_ABORTED	This is a reply for an operation or buffer that was aborted at the request of the host.
IOCSTATUS_TARGET_INVALID_DID	The Target Host attempted to send a request that indicated a D_ID value that is not in use.
IOCSTATUS_TARGET_FC_NODE_LOGGED_OUT	This operation has been canceled because the destination node has logged us out.

IOCLogInfo Should be logged by the host driver.

One can readily appreciate the many and various types kinds of events which could trigger the building of a reply message according to the method and program code of the invention. An address reply descriptor such as that depicted at 30 in FIG. 3A, is accordingly generated in order to locate the reply message to which it points. It is only when at least one condition is not met, indicating that the request was not processed without error, that such a reply message is generated for transmission over the I/O message passing medium. For example, occurrence of one or more of the following events, among others, could trigger the generation of a reply message: execution of any one command of the request is not completed initially or after a selected number of 'retrys', any one command of the request was made at an improper time, an allotted time for execution of any one command of the request was exceeded, unsuccessful data transfer of any portion of the request message, quantity of data transferred exceeded byte count specifications, quantity of data transferred was less-than that required in byte count specifications, processor resources were insufficient to execute any one command of the request, at least one field of the request message included invalid data, at least one value from the request message was out of range, data transferred was insufficient to execute any command of the request, hardware interface of the message passing medium was incompatible with the target device, communications protocol utilized by the message passing medium was incompatible with the target device, an unrecoverable bus parity error has occurred, a task management function has failed, a host processor aborted the request message, a target device node has logged-off, and so on.

The following is provided by way of reference: A message is any sized set or subset of data generated for transmission over a communications/message passing medium (which can comprise cabling, alone, or can include transmission through space from a transmitter to a receiver, or some combination thereof) whether local or physically remote. For each data element, there can be many fields in the database that hold the data items. As basic units of storage, a data element describes the logical unit of data (i.e., the logical definition of the field), fields are the physical storage units (typically one or more bytes in size), and data items are the individual instances of the data elements (i.e., actual data stored in the field). Computer readable storage medium/media, as used herein, can be any data carrier or recording medium into, or onto, which information (such as data) can be read and copied, such as magnetic (diskettes, hard disks, Iomega Corporation's ZIP™/JAZ™/Click!™ disks, tapes, drums, core, thin-film, etc.), optic (CD-ROM, CD-E, CD-R, CD-RW, DVD, and other devices whereby readout is with a light-source and photodetector), magneto-optic media (media for which optical properties can be changed by an applied magnetic field—used in high end drives), and other such devices.

A stack is a set of hardware registers or a reserved amount of memory used for arithmetic calculations, keep track of internal operations, etc. A CAM (Content Addressable Memory), also referred to as "associative storage", is storage that can be accessed by comparing the content of the data stored in it rather than by addressing predetermined locations. A buffer is any reserved segment of memory or accessible storage used to hold data during processing. A host can be any computer or computerized device that acts as a source of information or signals, including for example, a centralized mainframe that is a 'host' to its terminals, a server that is 'host' to its clients, to a desktop PC that is 'host' to its peripherals, and in network architectures, a client station that is a source of information to the network.

As mentioned above, the message passing medium over which reply descriptors may be transmitted may comprise

one or more parallel, serial, and wireless buses, r any hybrid thereof. More-specifically, suitable buses include those operational with a variety of hardware interface types such as those identified as SCSI; Fibre Channel; PCI (Peripheral Component Interconnect—commonly used to provide a high-speed data path between the CPU and peripheral devices such as video, disk, network, etc.); PCI-X; ISA (Industry Standard Architecture—an expansion bus commonly used in PCs along with ISA buses); InfiniBand; IDE (Integrated Drive Electronics—widely used to connect hard disks, CD-ROMs and tape drives to a PC); RS-232 (Recommended Standard-232—an TIA/EIA standard for serial transmission between computers and peripheral devices such as modem, mouse, etc., that uses a 25-pin DB-25 or 9-pin DB-9 connector); USB (Universal Serial Bus—used for low-speed peripherals such as the keyboard, mouse, joystick, scanner, printer and telephony devices); EISA (Extended ISA); Local Bus; Micro Channel; and so on.

The message passing medium can utilize any of a wide variety of suitable communications protocol such as SCSI; ATM (Asynchronous Transfer Mode—a network technology for both LANs, local area networks, and WANs, wide area networks); IPI (Intelligent Peripheral Interface-a high-speed hard disk interface used with minicomputers and mainframes that transfers data in the 10 to 25 MBytes/sec range); HiPPI (High Performance Parallel Interface—an ANSI-standard high-speed communications channel that uses a 32-bit or 64-bit cable and transmits at 100 or 200 Mbytes/sec); IP (Internet Protocol-the IP part of the TCP/IP communications protocol); InfiniBand, SSA (Serial Storage Architecture); IEEE P1394 (sometimes referred to as "FireWire"—high-speed serial bus communications that allows for the connection of up to 63 devices); and so on.

One can better appreciate the flexibility of the reply descriptor and message flow of the invention in connection with viewing FIG. 4 which, by way of example only, includes the novel intercommunication of two separate host drivers represented and labeled as #1 at 70 and #2 at 80. The transmission of information in the form of request and reply descriptors as well as request and reply messages throughout the I/O system represented can be accomplished through the use of, for example, a dual function in PCI (the logical separation of which is indicated by dashed line 85) and multiple service connections in InfiniBand. Steps detailing activity within a novel I/O system that can carry out a preferred method of the invention are summarized below in connection with corresponding data flow lines (labeled A through L in FIG. 4):

A. Each host driver 70, 80 initially receives an I/O request from an operating system (not shown, for simplicity).

B. Each host driver allocates a system message frame (SMF) from a collection of frames such as those shown at 71 and 81 which preferably resides in system/host memory, and builds an I/O request message within the SMF. Since, here, SMFs reside in host memory, the particular means by which allocation is carried out is the responsibility of the host driver, and can be one of any conventional SMF allocation method.

C. Each host driver 70, 80 generates its own request message frame descriptor (represented at 73, 83) for each respective I/O request message built by the host.

D. Each host driver writes its respective request message frame descriptor 73, 83 to a request queue labeled Request Post FIFO at 75 (in this novel example, the Request Post FIFO 75 manages request message frame descriptors from both host drivers 70, 80). At this point the IOC (represented at 90) takes ownership/control of the SMF within which each respective I/O request message resides.

E. The IOC 90 reads the request message frame descriptors from the Request Post FIFO 75 and DMA's (Direct Memory Access—whereby data is transferred from memory to memory without using the CPU) the request messages to a local message frame (here, 'local' to the IOC).

F. The IOC 90 generates and sends the appropriate request messages to a target device based on the port type associated with the Bus and TargetID fields of the request message. For an example of an I_2O request message structure which can be accommodated according to the invention—see I_2O Specification, section 3.4.1.2.1.

G. The IOC 90 receives the reply information from the target device—concerning whether the original request was processed without error, and if not, an indication of which condition was not met (the latter generally only to occur in a small fraction of the cases, such as for example, if a driver or interconnection is faulty).

H. If the I/O status is successful (request was processed without error) the IOC 90 writes data unique to the corresponding original request message, such as the MessageContext field value (see FIG. 3C at 38*b*) or other Protocol Dependent data (see FIG. 3B at 38*a* and FIG. 3D at 38*c*) to the respective reply queue (labeled 77, 87 Reply Post FIFO), in turn, causing an alert signal to be transmitted (e.g., a system interrupt is generated) letting the respective host driver 70, 80 know that a reply descriptor is in the Reply Post buffer 77, 87. At this point, the respective system/host driver (represented at 70, 80) regains ownership/control of the SMF within which each respective I/O request message resides.

I. If the I/O status is not successful (e.g., either the I/O request was not fully processed or it was processed but done so with an error, thus, the I/O request was not successfully completed), the IOC 90 removes an address to an available reply message frame from the reply queue (labeled 78, 88 Reply Free FIFO). The IOC 90 then generates a reply message in the host based message frame and writes the physical address of the reply message frame, shifted to the appropriate bits, to the respective reply queue (labeled 77, 87 Reply Post FIFO), in turn, causing a signal to be transmitted (e.g., a system interrupt is generated) informing the respective host driver 70, 80 that a reply descriptor is in the Reply Post buffer 77, 87. At this point, the respective system/host driver (represented at 70, 80) regains ownership/control of the SMF within which each respective I/O request message resides.

J. The system/host driver 70, 80 receives the respective interrupt (or other suitable signal) and reads the Reply Post FIFO buffer 77, 87 to get the content field (comprising data such as the MessageContext field value, see FIG. 3C at 38*b*, some other Protocol Dependent data, see FIG. 3B at 38*a* and FIG. 3D at 38*c*, or a PhysicalAddress, see FIG. 3A at 34, to the host-based reply message frame generated under step I. above). If there are no posted reply descriptors when the system/host driver 70, 80 reads a respective Reply Post FIFO 77, 87, the host driver 70, 80 will receive some arbitrary value (such as the value FFFFFFFFh) indicating that there are no more reply descriptors to be read.

K. Each respective host driver 70, 80 then responds to the initiator/caller (such as an operating system, not shown for simplicity) appropriately.

L. In the event a reply frame (72, 82) was needed during the process to respond to the initial I/O request (which generally occurs only in a very small fraction of the cases) and, thus, an available address was removed from the Reply Free buffer (78, 88) and a reply message generated and written to the allocated reply frame, the host driver (70, 80) returns the address to the respective Reply Free FIFO buffer 78, 88.

Referring back once more to FIGS. 3A and 3B to summarize the features depicted by FIG. 4: The Address Reply mechanism (such as that explained above as step I. where the I/O status is not successful) requires an IOC 90 to remove an available SMF address from the Reply Free buffer (FIG. 4 at 78 or 88), build a reply message, DMA the reply message to the respective host driver 70, 80, as well as build a reply descriptor (FIG. 3A at 30) and place it on a respective Reply Post buffer (FIG. 4 at 77, 87). On the other hand, the much more efficient context reply mechanism of the invention (explained above as step H. where the I/O status is successful) reduces these PCI accesses, thus increasing performance, to a single write of a reply descriptor (such as any of those depicted in FIGS. 3B, 3C, and 3D at 35, 40, and 45) to the Reply Post buffer (FIG. 4 at 77, 87). This is accomplished by having the IOC write a single content field (of any designated length) shifted the appropriate bits to accommodate an indication field (that can function to identify the general type of reply descriptor, see FIGS. 3B, 3C, and 3D at 36*a*, 36*b*, and 36*c*) to a Reply Post buffer (FIG. 4 at 77, 87). See also, the data flow diagram labeled FIG. 5.

Therefore, as one can readily appreciate that the novel context reply mechanism of the invention requires that a host driver do only a single PCI Read to retrieve the context and indication fields (such as are referenced at 38*a*–38*c* and 36*a*–36*c* of reply descriptors 35, 40, 45, respectively) from a Reply Post buffer, and no PCI Write to post the Reply Free SMF address is needed in the majority of instances where no reply message frame is used when the I/O status is successful. Note that the figures label several queues specifically as FIFO (first-in-first-out), although they need not be that particular type of buffer.

Turning to the data flow diagram labeled FIG. 5, identified (for reference only) along the left hand side (see arrow 120) are blocks representing the various logically defined layers of an I/O system execution environment. Details of the I/O system data flow represented in FIG. 5 are set forth below in connection with reference numbers:

101. An initiator/caller Operating System (OS) issues a request or command.

102. An OSM (operating system module, or OS Specific Device Driver)—such as the two modules referenced in FIG. 4 as host drivers 70, 80—accepts the request and translates it into a request message addressed to a target DDM. The OSM has the option to place a pointer (location indicator, such as a physical or virtual address) to the OS I/O request in the request message's TransactionContext field.

103. The OSM (system/host driver e.g., the host drivers at 70, 80 in FIG. 4) invokes the communication layer to deliver the request message.

104. The host driver queues the request message (can be done via mechanism such as that identified in I_2O Specification as MessengerInstance) by copying it into a message frame buffer residing on an IOC.

105. The IOC posts the request message to the target DDM's event queue.

106. The target DDM processes (or, at least attempts to process) the request.

107. After the target DDM processes and satisfies the request successfully (which occurs for the vast majority of request messages transmitted), the IOC generates a reply descriptor for that request. Note here that, although not specifically illustrated in FIG. 5, for the small percentage of request messages that are not 'processed without error' an available reply message frame address is removed from available message frame addresses residing in a buffer (e.g., Reply Free buffer at 78 or 88 in FIG. 4), a reply message is built by the IOC to include information concerning the fault, error, failure, etc., (i.e., that which has not been met) and copied into the reply message frame. The IOC then generates a reply descriptor (with an address, or whereabouts, of the reply message frame, with bits shifted to accommodate an indication field).

108. The reply descriptor generated (whether it be an Address Reply or the more often encountered Context Reply, see FIGS. 3A–3D) is queued to a Reply Post buffer (such as those shown in FIG. 4 at 77, 87).

109. The writing of the reply descriptor to a Reply Post buffer can be done in connection with some type of an alert signal, e.g., a system interrupt, to the system/host driver that a reply descriptor is ready for reading. The system driver receives the alert signal and reads the Reply Post buffer to get the reply descriptor information. Due to indication field 32 of the reply descriptor (see examples in FIGS. 3A–3D), it is readily apparent whether the descriptor is of an Address or Context reply type. If the descriptor is of the Address type, the associated reply message must be found and read.

110. The system/host driver then correlates each specific reply response with an original, corresponding request to complete the response process. For example, the host driver can retrieve an address-pointer to the original I/O request by looking at a Context reply descriptor's context field (see 38*a*, 38*b*, 38*c* in FIGS. 3B–3D) or looking at an I/O request identifier found in any reply message, if it was necessary to so generate a reply message (see FIG. 8B for an example format).

111. The host driver returns the I/O request to the initiator operating system.

Since a host system generally communicates with an IOC through the use of System Interface registers, by way of example only, FIG. 6 illustrates an example System Interface Register Map 130 for this purpose. Access to registers can be provided as is customary in I/O architecture, via memory and/or I/O mapping.

FIG. 7A illustrates yet another example of a request message format, namely a Config request message 140, for supporting configuration operations such as read/write. The format of an example reply message to the FIG. 7A request, is represented at 150 in FIG. 7B. The Config request message 140, such as that in FIG. 7A, is used to access operational parameters supported by the IOC.

Description of Fields Represented in the Example Config Reply Message of FIG. 7B

IOCStatus	This field is used to return IOC supplied information specific to Configuration requests in addition to the function independent values specified as shown in the Default Reply Message.

IOCStatus	Description
IOCSTATUS_CONFIG_BAD_ACTION	The action is not supported
IOCSTATUS_CONFIG_BAD_TYPE	The configuration type is not supported
IOCSTATUS_CONFIG_BAD_PAGE	The configuration page is not supported
IOCSTATUS_CONFIG_BAD_DATA	Incorrect field setting within the configuration data
IOCSTATUS_CONFIG_NO_DEFAULTS	Can not set defaults for this page
IOCSTATUS_CONFIG_CANT_COMMIT	Non-volatile memory not available or error while writing persistent data to non-volatile memory
Reserved	Function specific fields.

FIGS. 8A and 8B depict an alternative SCSI request message format labeled **160** and corresponding error reply message format **170**: SCSI Initiator IO message represented in FIG. 8A is used to send a specific class of requests, namely, SCSI I/O requests to specific target devices. And according to the novel features of the invention, the SCSI IO Error reply message **170** in FIG. 8B need only be generated and transmitted if the SCSI IO request experiences any of a number of errors or failures during the process to carry it out (e.g., an event occurs such that any one of a set of predefined conditions is not met) and the IO request is, therefore, not completed.

Description of Fields Represented in the Example SCSI IO Request Message of FIG. 8A

TargetID	The target device identification number.
Bus	The SCSI bus number that the target device exists on.
CDBLength	Number of used bytes in CDB field.
MessageFlags	All reserved bits must have a value of 0.
LUN	The Logical Unit Number of the target device.
CDB	SCSI Command Descriptor Block.

-continued

SGL	The scatter gather list which identifies the memory location of the data for this IO.

Description of Additional Fields Represented in the FIG. 8B Example Error Reply Message

IOCLogInfo	An implementation specific value intended to supplement the IO Controller status.
IOCStatus	This field is used to return IOC supplied information specific to SCSI IO requests in addition to the function independent values specified in FIG. 3F Default Reply Message.

IOCStatus examples	Description
IOCSTATUS_SCSI_RECOVERED_ERROR	I/O operation completed successfully after retries
IOCSTATUS_SCSI_INVALID_BUS	Out of range Bus value in request message
IOCSTATUS_SCSI_INVALID_TARGETID	Out of range TargetID value in request message
IOCSTATUS_SCSI_DEVICE_NOT_THERE	Selection time-out or device does not exist
IOCSTATUS_SCSI_DATA_OVERRUN	SCSI device attempted to transfer more data than the amount specified by the byte count
IOCSTATUS_SCSI_DATA_UNDERRUN	SCSI device transferred less data than the amount specified by the byte count
IOCSTATUS_SCSI_IO_DATA_ERROR	I/O terminated because of unrecoverable bus parity CRC error
IOCSTATUS_SCSI_PROTOCOL_ERROR	I/O terminated because of unrecoverable bus protocol error
IOCSTATUS_SCSI_TASK_TERMINATED	I/O terminated because of SCSI Task Management Request
IOCSTATUS_SCSI_BUS_RESET	I/O terminated because of a Bus Reset unrelated to a SCSI Task Management Request
IOCSTATUS_SCSI_TASK_MGMT_FAILED	SCSI Task Management function failed
Reserved	Function dependent fields.

While certain representative embodiments and details have been shown merely for the purpose of illustrating the invention, those skilled in the art will readily appreciate that various modifications may be made without departing from the novel teachings or scope of this invention. Accordingly, all such modifications are intended to be included within the scope of this invention as defined in the following claims. Although the commonly employed preamble phrase "comprising the steps of" may be used herein, or hereafter, in a method claim, the Applicants in no way intends to invoke 35 U.S.C. section 112 ¶6. Furthermore, in any claim that is filed hereafter, any means-plus-function clauses used, or later found to be present, are intended to cover the structures described herein as performing the recited function and not only structural equivalents but also equivalent structures.

What is claimed is:

1. A reply descriptor for transmission over an I/O message passing medium in response to a corresponding request message, comprising:

at least one indication field that identifies type of the reply descriptor, and a content field; and

whereby a reply message is generated only if at least one predefined condition is not met and said content field comprises information of said reply message's storage location, if so generated.

2. The reply descriptor of claim 1 wherein:

the message passing medium comprises a bus operational with a hardware interface type selected from the group consisting of SCSI (Small Computer System Interface), Fibre Channel, PCI (Peripheral Component Interconnect), PCI-X, ISA (Industry Standard Architecture), InfiniBand, IDE (Integrated Drive Electronics), USB (Universal Serial Bus), RS-232, EISA (Extended ISA), Local Bus, and Micro Channel; and

the message passing medium utilizes a communications protocol selected from the group consisting of SCSI, ATM (Asynchronous Transfer Mode), IPI (Intelligent Peripheral Interface), HiPPI (HIgh Performance Parallel Interface), IP (Internet Protocol), InfiniBand, SSA (Serial Storage Architecture), and IEEE P1394.

3. The reply descriptor of claim 1 wherein: upon the writing of the reply descriptor to a reply-post buffer, an interrupt is transmitted for a host-based driver to read the reply descriptor; and once so read, said host-based driver correlates the reply descriptor with the request message and sends a notification message to an originating-caller.

4. The reply descriptor of claim 1 wherein:

said indication field comprises an address bit;

if each said predefined condition is met, said indication field further comprises a type field, and said content field comprises data copied from and unique to the request message as generated by a host-based driver; but

if said reply message is so generated, said reply message comprises data regarding said at least one predefined condition not met.

5. The reply descriptor of claim 4 wherein:

if each said predefined condition is met, said content field further comprises a receiving port identifier; and said data unique to the request message comprises an identifier selected from the group consisting of: an address to a storage space in a memory, an index value to a table, an index value to a list, an index value to a register, an index value to a stack, an index value to an array, and content-data associated with a hardware assisted CAM; and

said reply-post buffer is a FIFO (First-In-First-Out) type buffer.

6. A reply descriptor for transmission over an I/O message passing medium in response to a corresponding request message, comprising:

at least one indication field comprising an address bit, and a content field;

whereby a reply message is generated only if at least one predefined condition is not met and said content field comprises information of said reply message's storage location, if so generated;

wherein each said predefined condition is met: said indication field further comprises a type field, said content field comprises data copied from and unique to the request message as generated by a host-based driver, said data unique to the request message comprises a host-specified index value;

an alert signal for said host-based driver is transmitted upon the writing of both the reply descriptor and a second reply descriptor to a reply-post buffer, said second reply descriptor comprising a second content field having second data copied from a second request message, and once said reply descriptors have been read by said host-based driver, said host-based driver correlates each said reply descriptor with a respective request message and notifies an originating-caller of completion of both of said request messages.

7. The reply descriptor of claim 1 wherein: said at least one predefined condition is not met and said content field comprises said information, said information to comprise an address to an available reply frame buffer located in a host memory; and said address is one from a plurality of addresses, each of which identifies a location of a corresponding reply frame buffer.

8. The reply descriptor of claim 7 wherein said at least one predefined condition is not met because execution of at least one command included in the request message was not completed, said corresponding reply frame buffers reside in said host memory, and said plurality of addresses resides on a reply-free FIFO buffer.

9. The reply descriptor of claim 8 wherein:

once said address to said available reply frame buffer is removed from said reply-free FIFO buffer, said reply message is generated by an I/O controller (IOC) and copied into said available reply frame buffer; and

the reply descriptor is written to a reply-post buffer for a host-based driver to read.

10. The reply descriptor of claim 1 wherein:

the request message comprises at least one command; and

said at least one predefined condition is not met upon the occurrence of an event selected from the group consisting of: execution of said command is not completed, execution of said command is not completed after a retry, said command is made at an improper time, an allotted time for execution of said command is exceeded, unsuccessful data transfer of any portion of the request message, quantity of data transferred exceeds byte count specifications, quantity of data transferred is less-than byte count specifications, processor resources are insufficient to execute said command, at least one field of the request message comprises invalid data, at least one value from the request message is out of range, any data transferred is insufficient to execute said command, hardware interface of the message passing medium is incompatible with a target device, communications protocol utilized by the message passing medium is incompatible with a target device, an unrecoverable bus parity error has occurred, a task management function has failed, a host processor aborts the request message, and a target device node has logged-off.

11. A method of responding over an I/O message passing medium, to a request message, the method comprising the steps of:

generating a reply message to the request message only if at least one predefined condition is not met;

generating a reply descriptor having at least one indication field that identifies type of said reply descriptor, and a content field; whereby said content field com-

prises information of said reply message's storage location if said reply message is so generated.

12. The method of claim 11 wherein: if each said predefined condition is met, said content field to comprise data copied from the request message rather than said storage location information.

13. The method of claim 12 wherein:

the message passing medium comprises a bus operational with a hardware interface type selected from the group consisting of SCSI (Small Computer System Interface), Fibre Channel, PCI (Peripheral Component Interconnect), PCI-X, ISA (Industry Standard Architecture), InfiniBand, IDE (Integrated Drive Electronics), USB (Universal Serial Bus), RS-232, EISA (Extended ISA), Local Bus, and Micro Channel; and

the message passing medium utilizes a communications protocol selected from the group consisting of SCSI, ATM (Asynchronous Transfer Mode), IPI (Intelligent Peripheral Interface), HiPPI (HIgh Performance Parallel Interface), IP (Internet Protocol), InfiniBand, SSA (Serial Storage Architecture), and IEEE P1394.

14. The method of claim 12 wherein said indication field comprises an address bit, and further comprising the steps of:

if each said predefined condition is met, generating said indication field to further comprise a type field and generating said data copied to comprise data unique to the request message; but

said at least one predefined condition is not met, generating said reply message to comprise data regarding said at least one predefined condition not met.

15. The method of claim 14 further comprising the steps of:

upon the writing of said reply descriptor to a reply-post buffer, transmitting an alert signal for said host-based driver to read said reply descriptor; and

once so read, correlating said reply descriptor with the request message and notifying an originating-caller of a completion results.

16. The method of claim 12 further comprising the steps of initially receiving an I/O request comprising at least one command from an operating system, and generating the request message to include said one command; and wherein said at least one predefined condition is not met upon occurrence of an event selected from the group consisting of: execution of said command is not completed, execution of said command is not completed after a retry, said command is made at an improper time, an allotted time for execution of said command is exceeded, unsuccessful data transfer of any portion of the request message, quantity of data transferred exceeds byte count specifications, quantity of data transferred is less-than byte count specifications, processor resources are insufficient to execute said command, at least one field of the request message comprises invalid data, at least one value from the request message is out of range, any data transferred is insufficient to execute said command, hardware interface of the message passing medium is incompatible with a target device, communications protocol utilized by the message passing medium is incompatible with a target device, an unrecoverable bus parity error has occurred, a task management function has failed, a host processor aborts the request message, and a target device node has logged-off.

17. The method of claim 12 wherein said predefined condition is met, and said step of generating said reply descriptor further comprises writing said content field to a reply-post buffer, said content field to further comprise a receiving port identifier and a request-initiator identifier; and further comprising the step of transmitting a system interrupt for a host-based driver to read said reply descriptor.

18. The method of claim 17 further comprising the steps of: reading said reply descriptor; correlating a request message identifier of said reply descriptor with the request message; and notifying an originating-caller to confirm completion of the request message.

19. The method of claim 17 wherein said step of generating said reply descriptor is performed with an I/O controller (IOC); and said step of transmitting a system interrupt is performed with said IOC; and further comprising a reply queue register located in an IOC memory, said register to comprise said reply-post buffer and a reply free buffer on which a plurality of addresses, each of which identifies a location of a corresponding reply frame buffer, reside.

20. The method of claim 12 wherein said at least one predefined condition is not met, and said step of generating said reply descriptor further comprises writing said content field to a reply-post buffer, said content field to further comprise said information including an address to an available reply frame buffer located in a host memory, said address having been taken from a reply-free on which a plurality of addresses also reside; and further comprising the step of: generating said reply message with said IOC and copying said reply message into said available reply frame buffer.

21. A computer executable program code on a computer readable storage medium, the program code comprising:

a first program sub-code for generating a reply message to a corresponding I/O request message only if at least one predefined condition is not met; and

said first program sub-code comprising instructions for generating a reply descriptor having at least one indication field and a content field that comprises information of said reply message's storage location if said reply message is so generated, but said content field to comprise data copied from said I/O request message if each said predefined condition is met.

22. The program code of claim 21 wherein said instructions for generating said reply descriptor further comprise instructions for writing said content field to a reply-post buffer, and further comprising:

a second sub-code for transmitting a system interrupt over an I/O message passing medium for a host-based driver to read said reply descriptor;

a third sub-code for reading said reply descriptor; and

a fourth sub-code for correlating said reply descriptor with said corresponding I/O request message and notifying an originating-caller of a completion results.

23. The program code of claim 22 wherein each said predefined condition is met and said content field comprises data copied from said I/O request message including an identifier selected from the group consisting of: an address to a storage space in a memory, an index value to a table, an index value to a list, an index value to a register, an index value to a stack, an index value to an array, and content-data associated with a hardware assisted CAM; said third sub-code comprises instructions for reading into a host-based memory; and said fourth program sub-code comprises instructions for correlating said request message identifier of said reply descriptor with the I/O request message.

24. The program code of claim 22 wherein said at least one predefined condition is not met because execution of at least one command included in said I/O request message was not completed and said content field comprises said information, said information to comprise an address to an available reply frame buffer located in a host memory; and said address is one from a plurality of addresses, each of which identifies a location of a corresponding reply frame buffer residing in said host memory.

25. The program code of claim 24 wherein said third sub-code comprises instructions for reading said reply descriptor from a reply-post buffer.

* * * * *

US006085265A

United States Patent [19]

Kou

[11] Patent Number: 6,085,265

[45] Date of Patent: Jul. 4, 2000

[54] SYSTEM FOR HANDLING AN ASYNCHRONOUS INTERRUPT A UNIVERSAL SERIAL BUS DEVICE

[75] Inventor: James Tai-Ling Kou, Corona, Calif.

[73] Assignee: Toshiba America Information Systems, Inc., Irvine, Calif.

[21] Appl. No.: 09/004,835

[22] Filed: Jan. 9, 1998

[51] Int. Cl.[7] G06F 3/00; G06F 13/00

[52] U.S. Cl. 710/63; 710/1; 709/301; 370/276; 714/1

[58] Field of Search 710/1, 8, 63, 260, 710/129, 126, 27, 266, 100; 709/301; 714/1; 370/465, 276; 361/683; 439/604

[56] References Cited

U.S. PATENT DOCUMENTS

5,291,609	3/1994	Herz	710/52
5,499,384	3/1996	Lentz et al.	710/1
5,566,346	10/1996	Andert et al.	710/8
5,615,404	3/1997	Knoll et al.	710/62
5,621,898	4/1997	Wooten	710/117
5,630,141	5/1997	Ross et al.	710/261
5,636,211	6/1997	Newlin et al.	370/465
5,649,129	7/1997	Kowert	710/129
5,664,126	9/1997	Hirakawa et al.	345/329
5,784,581	7/1998	Hannah	710/110
5,822,182	10/1998	Scholder et al.	361/683
5,831,597	11/1998	West et al.	345/163
5,845,151	12/1998	Story et al.	710/27
5,859,993	1/1999	Snyder	712/208

OTHER PUBLICATIONS

Internet Page: //www.microsoft.co.za/hwdev/pcfuture/wd-musb.HTM.

Intel Corporation/Microsoft Corporation: Advanced Power Management (APM) BIOS Interface Specification; Revision 1.2; Feb. 1996.

ComputerBoards, Inc.; PCI-GPIB; vol. 18, Sep. 1997.

Universal Serial Bus Specification; Revision 1.0, Jan. 15, 1996.

Universal Serial Bus (USB); Device Class Definition for Human Interface Devices (HID); Firmware Specification —Released Sep., 1996, Version 1.0 draft.

Primary Examiner—Gopal C. Ray

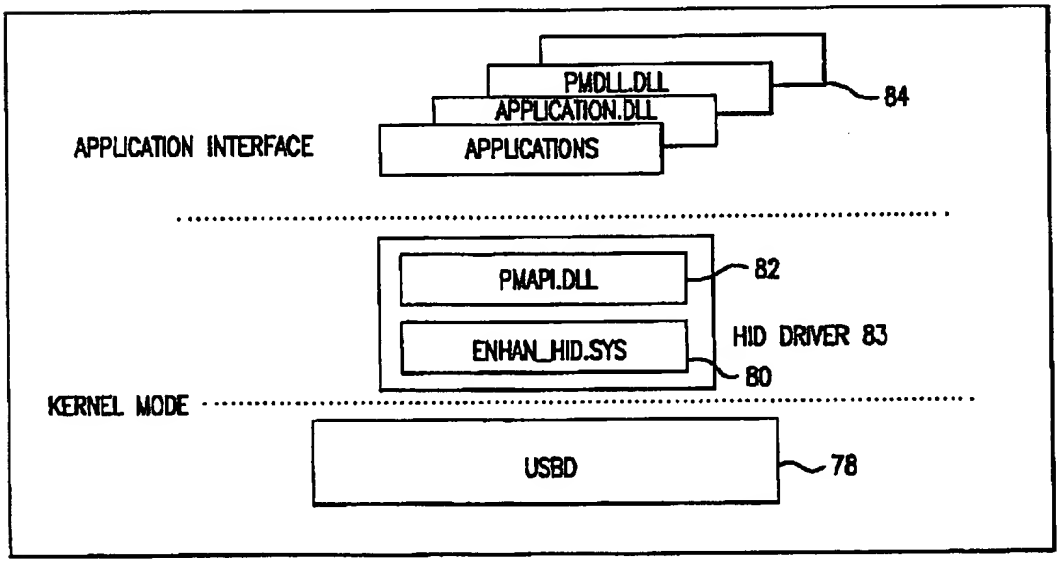
Attorney, Agent, or Firm—Pillsbury, Madison & Sutro LLP

[57] ABSTRACT

A system and method for establishing communication between a host computer and a peripheral device. The host computer includes logic for associating an attached peripheral device with one of a particular peripheral device type, logic for associating the attached peripheral device with an instance of a software driver executable on the processor, and logic for enabling communication between the attached device and the host after a set period of time after detection of the attachment of the USB device to the at least one port. The software driver is preferably capable of supporting communication between peripheral devices of the associated particular type and the host.

26 Claims, 10 Drawing Sheets

Microfiche Appendix Included
(2 Microfiche, 121 Pages)



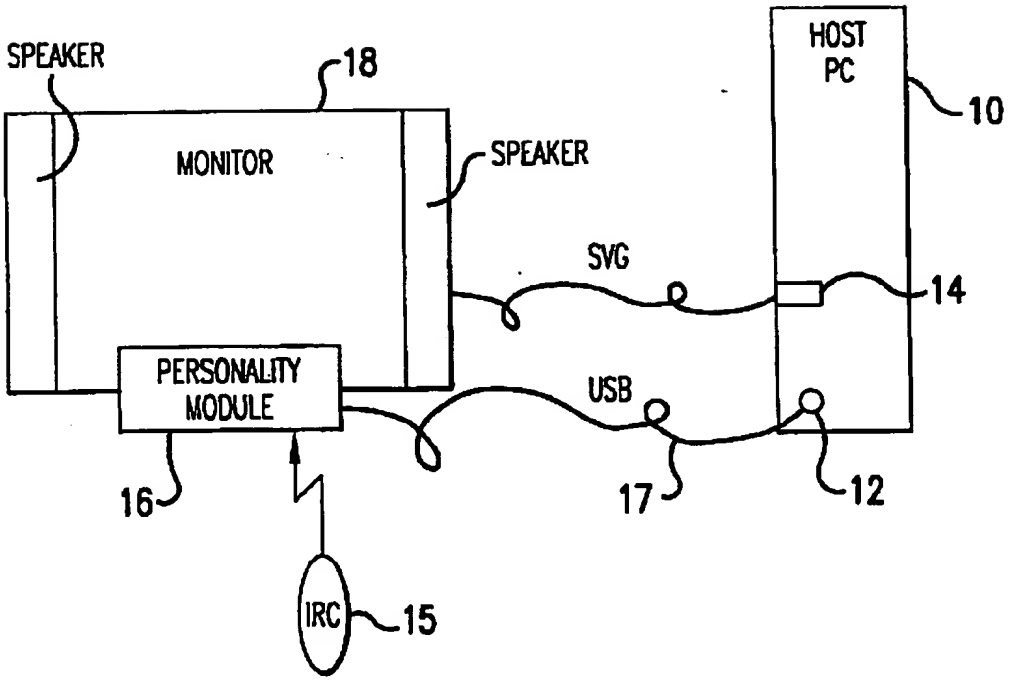

SPEAKER
18
HOST
PC
10
MONITOR
SPEAKER
SVG
14
PERSONALITY
MODULE
USB
16
17
12
IRC
15

FIG.1

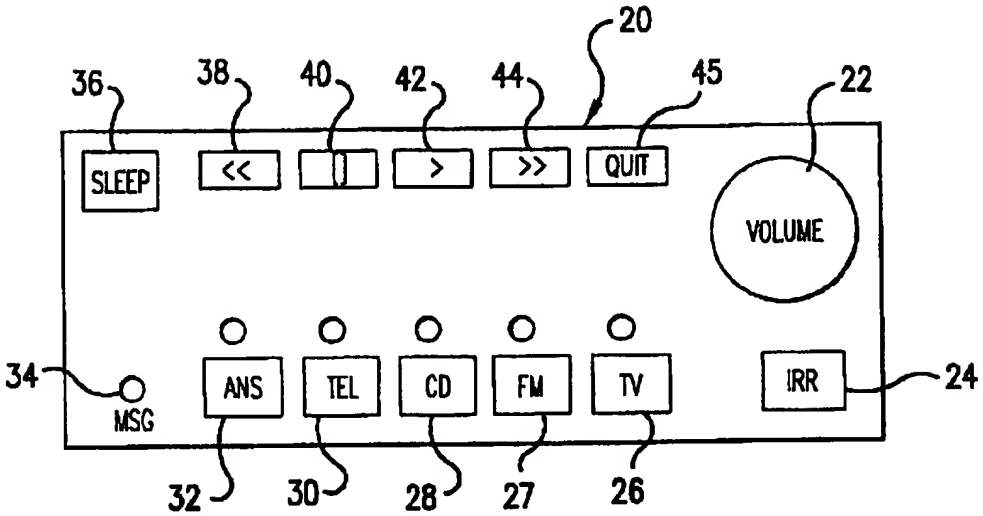

20
36
38
40
42
44
45
22
SLEEP
<<
>
>>
QUIT
VOLUME
34
MSG
ANS
TEL
CD
FM
TV
IRR
24
32
30
28
27
26

FIG.2

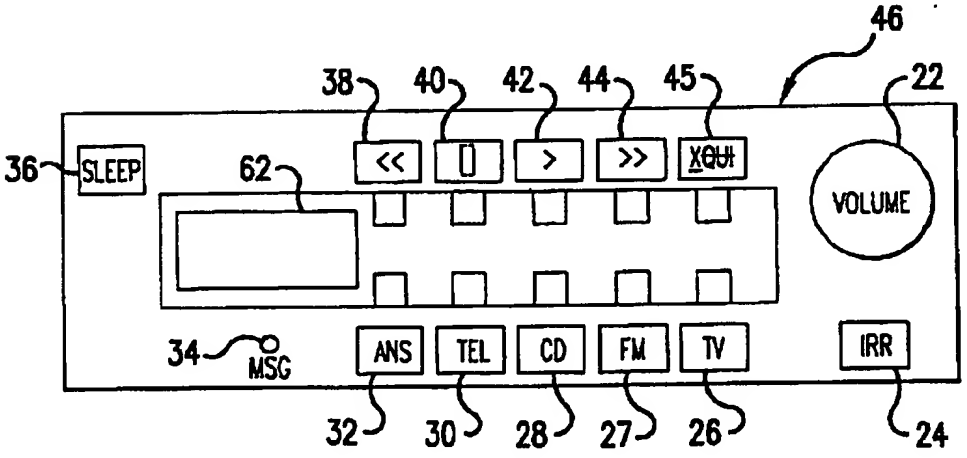

46
38
40
42
44
45
22
36
SLEEP
62
<<
>
>>
XOUT
VOLUME
34
MSG
ANS
TEL
CD
FM
TV
IRR
32
30
28
27
26
24

FIG.3

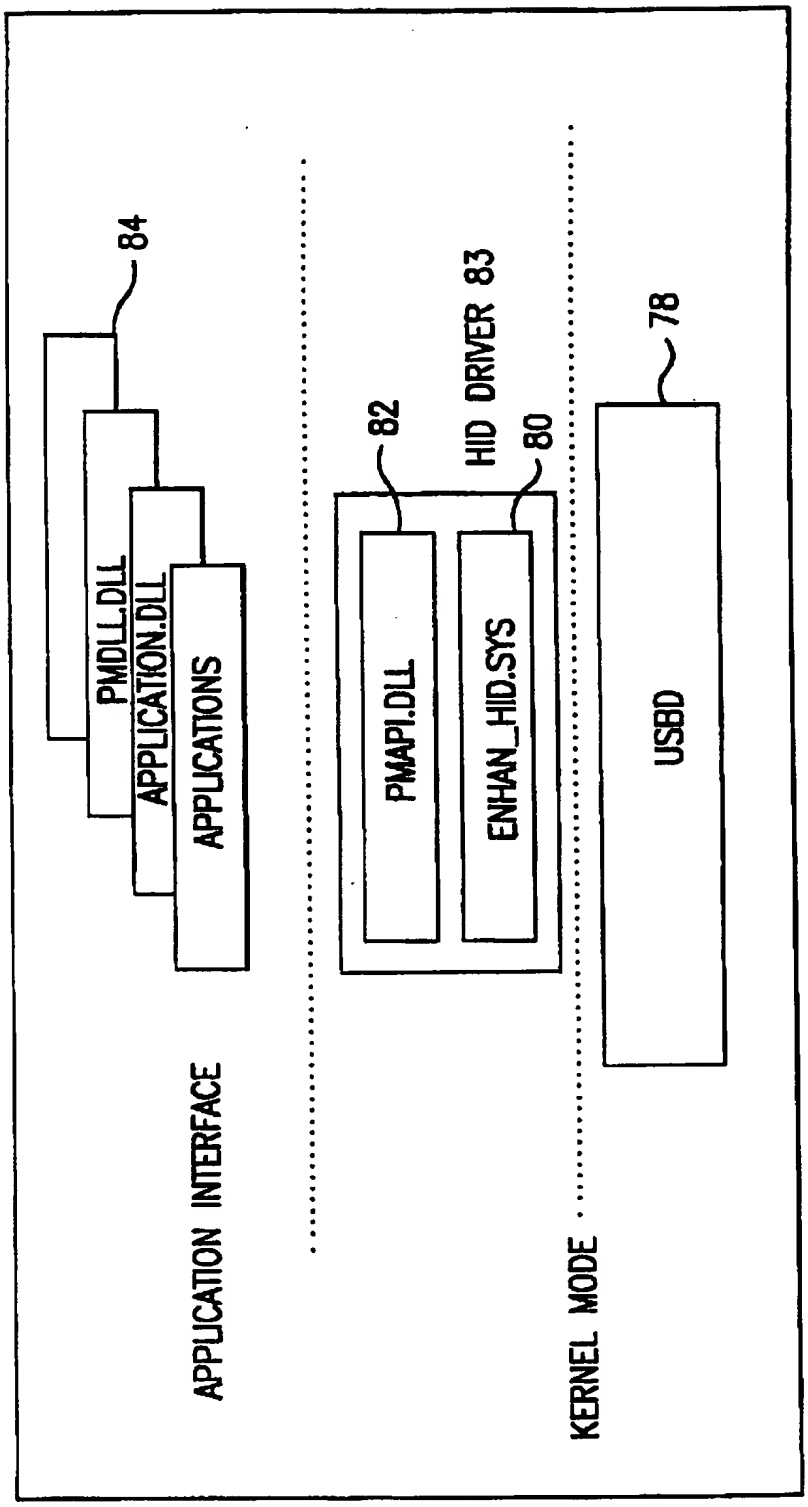

84
PMDLL.DLL
APPLICATION.DLL
APPLICATIONS
APPLICATION INTERFACE
82
HID DRIVER 83
80
PMAPI.DLL
ENHAN_HID.SYS
78
USBD
KERNEL MODE

FIG.4

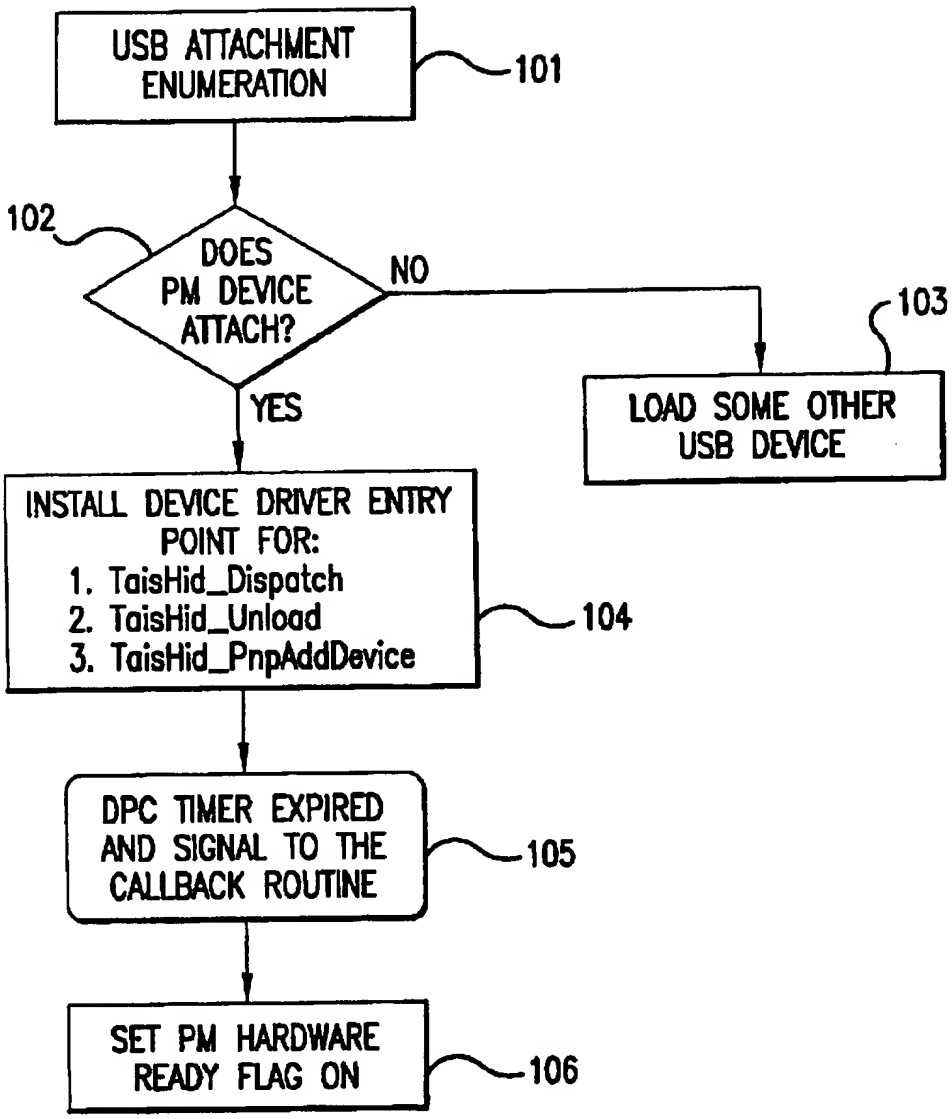

USB ATTACHMENT ENUMERATION
101
102
DOES PM DEVICE ATTACH?
NO
YES
103
LOAD SOME OTHER USB DEVICE
INSTALL DEVICE DRIVER ENTRY POINT FOR:
1. TaisHid_Dispatch
2. TaisHid_Unload
3. TaisHid_PnpAddDevice
104
DPC TIMER EXPIRED AND SIGNAL TO THE CALLBACK ROUTINE
105
SET PM HARDWARE READY FLAG ON
106

FIG.5

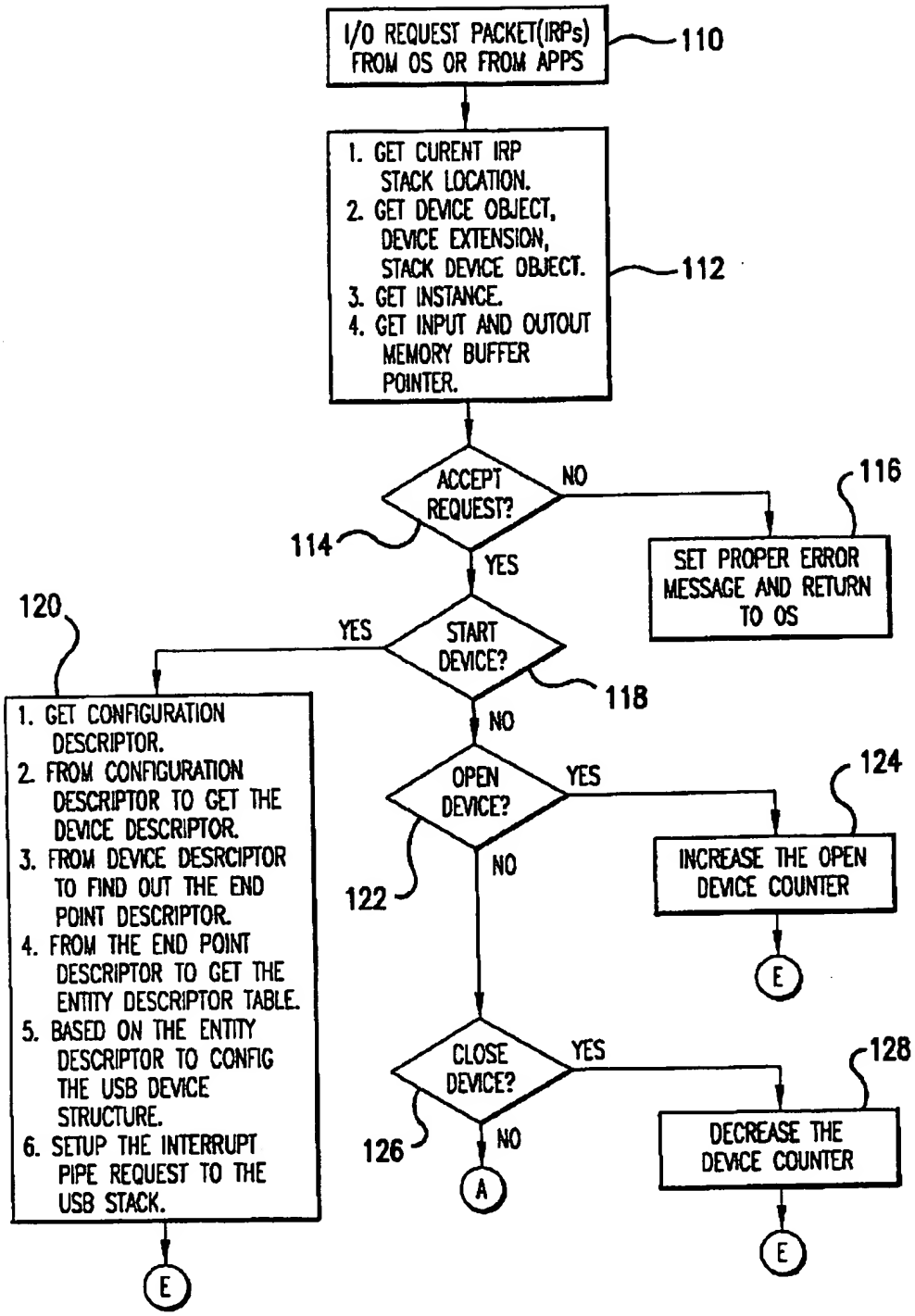

I/O REQUEST PACKET(IRPs) FROM OS OR FROM APPS
110
1. GET CURENT IRP STACK LOCATION.
2. GET DEVICE OBJECT, DEVICE EXTENSION, STACK DEVICE OBJECT.
3. GET INSTANCE.
4. GET INPUT AND OUTOUT MEMORY BUFFER POINTER.
112
ACCEPT REQUEST?
NO
116
114
YES
SET PROPER ERROR MESSAGE AND RETURN TO OS
120
YES
START DEVICE?
118
NO
1. GET CONFIGURATION DESCRIPTOR.
2. FROM CONFIGURATION DESCRIPTOR TO GET THE DEVICE DESCRIPTOR.
3. FROM DEVICE DESRCIPTOR TO FIND OUT THE END POINT DESCRIPTOR.
4. FROM THE END POINT DESCRIPTOR TO GET THE ENTITY DESCRIPTOR TABLE.
5. BASED ON THE ENTITY DESCRIPTOR TO CONFIG THE USB DEVICE STRUCTURE.
6. SETUP THE INTERRUPT PIPE REQUEST TO THE USB STACK.
OPEN DEVICE?
YES
124
122
NO
INCREASE THE OPEN DEVICE COUNTER
E
CLOSE DEVICE?
YES
128
126
NO
DECREASE THE DEVICE COUNTER
A
E
E

FIG. 6a

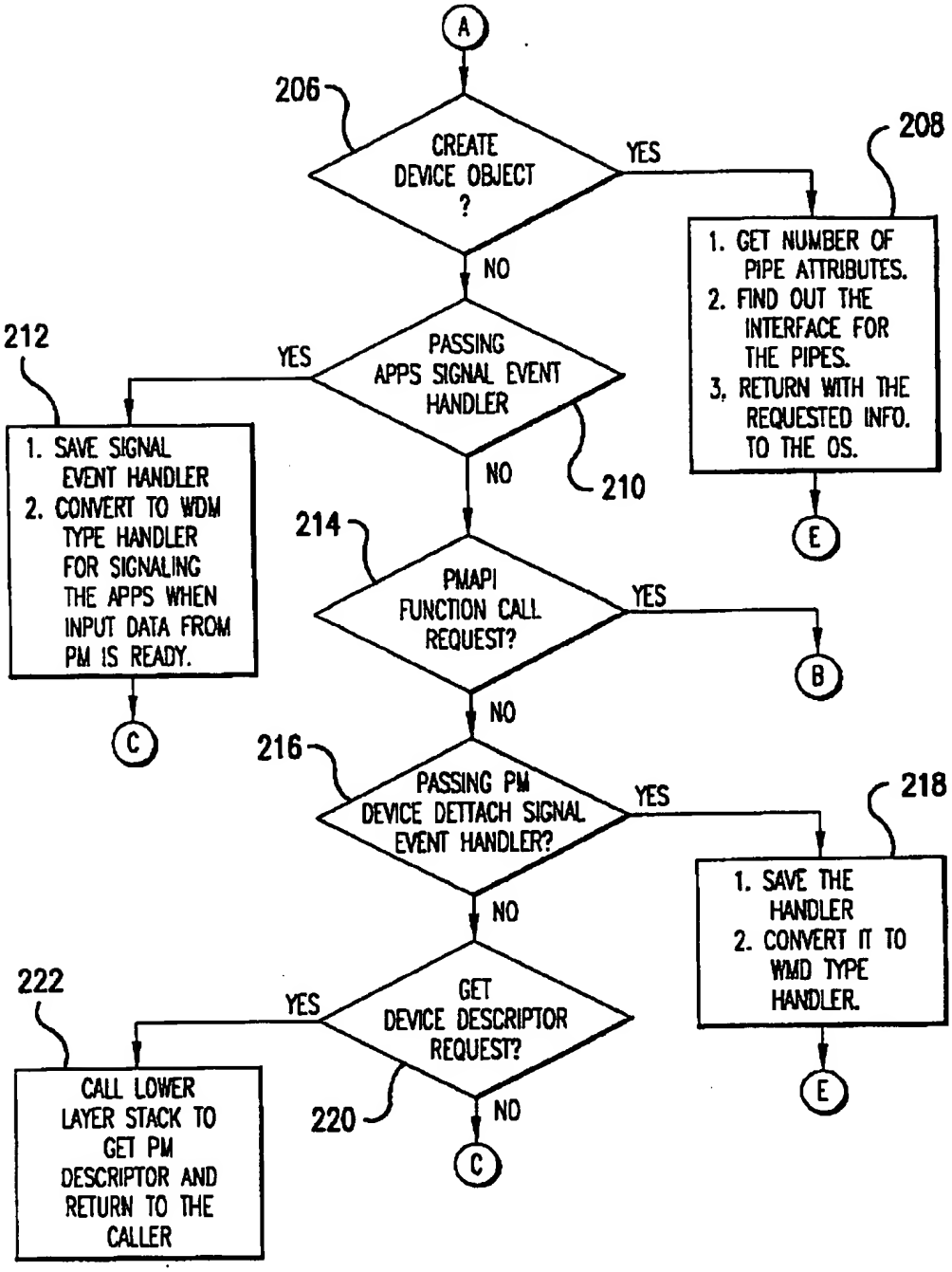

A
206
CREATE DEVICE OBJECT ?
YES
208
1. GET NUMBER OF PIPE ATTRIBUTES.
2. FIND OUT THE INTERFACE FOR THE PIPES.
3. RETURN WITH THE REQUESTED INFO. TO THE OS.
E
NO
212
PASSING APPS SIGNAL EVENT HANDLER
YES
210
NO
1. SAVE SIGNAL EVENT HANDLER
2. CONVERT TO WDM TYPE HANDLER FOR SIGNALING THE APPS WHEN INPUT DATA FROM PM IS READY.
C
214
PMAPI FUNCTION CALL REQUEST?
YES
B
NO
216
PASSING PM DEVICE DETTACH SIGNAL EVENT HANDLER?
YES
218
1. SAVE THE HANDLER
2. CONVERT IT TO WMD TYPE HANDLER.
E
NO
222
GET DEVICE DESCRIPTOR REQUEST?
YES
CALL LOWER LAYER STACK TO GET PM DESCRIPTOR AND RETURN TO THE CALLER
220
NO
C

FIG. 6b

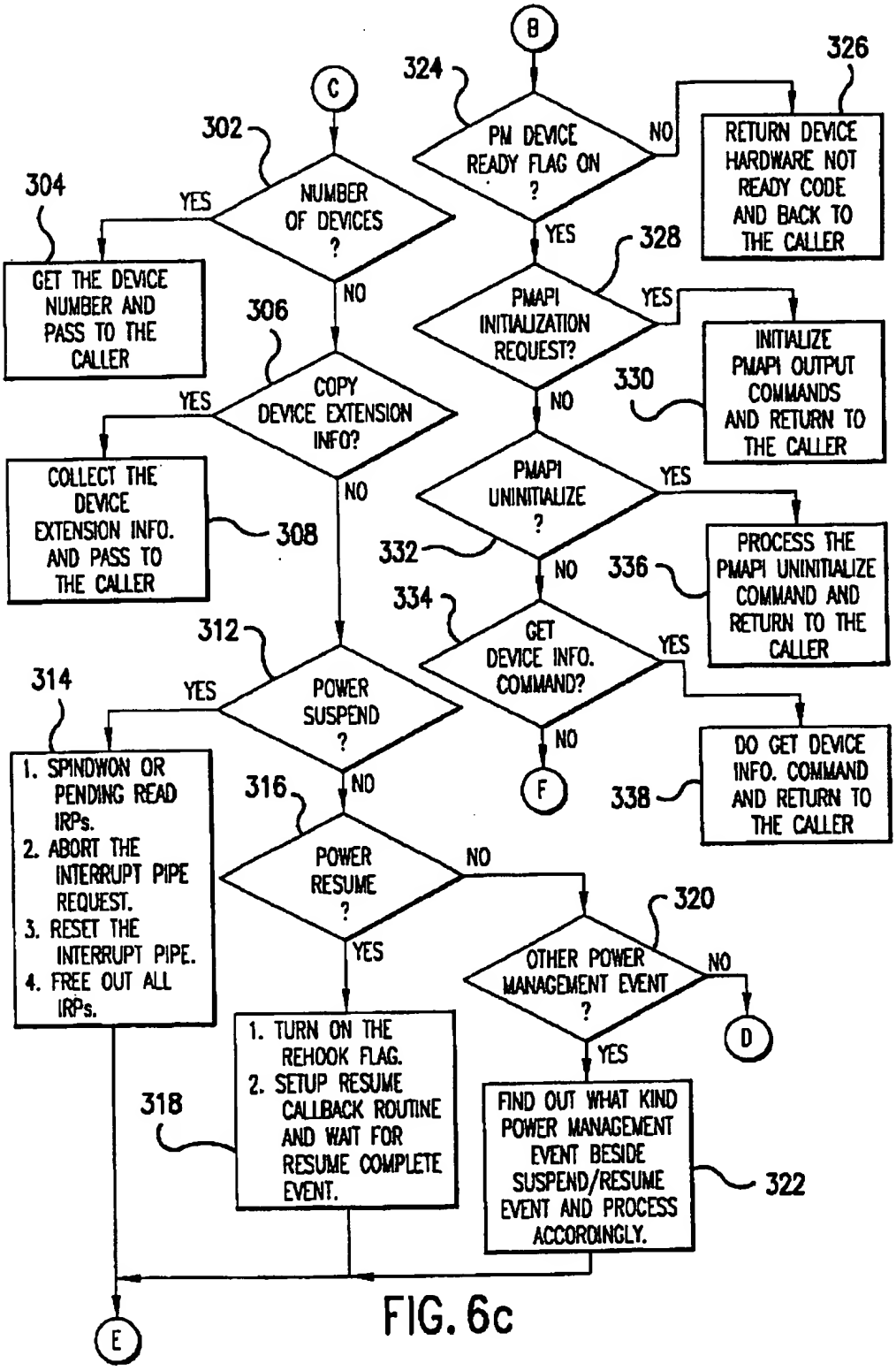

B
C
324
326
302
PM DEVICE READY FLAG ON ?
NO
RETURN DEVICE HARDWARE NOT READY CODE AND BACK TO THE CALLER
304
YES
NUMBER OF DEVICES ?
YES
328
GET THE DEVICE NUMBER AND PASS TO THE CALLER
NO
PMAPI INITIALIZATION REQUEST?
YES
306
INITIALIZE PMAPI OUTPUT COMMANDS AND RETURN TO THE CALLER
330
COPY DEVICE EXTENSION INFO?
YES
NO
PMAPI UNINITIALIZE ?
YES
COLLECT THE DEVICE EXTENSION INFO. AND PASS TO THE CALLER
308
332
NO
336
PROCESS THE PMAPI UNINITIALIZE COMMAND AND RETURN TO THE CALLER
334
GET DEVICE INFO. COMMAND?
YES
312
POWER SUSPEND ?
YES
314
NO
NO
F
338
DO GET DEVICE INFO. COMMAND AND RETURN TO THE CALLER
1. SPINDWON OR PENDING READ IRPs.
2. ABORT THE INTERRUPT PIPE REQUEST.
3. RESET THE INTERRUPT PIPE.
4. FREE OUT ALL IRPs.
316
POWER RESUME ?
NO
320
YES
OTHER POWER MANAGEMENT EVENT ?
NO
D
YES
1. TURN ON THE REHOOK FLAG.
2. SETUP RESUME CALLBACK ROUTINE AND WAIT FOR RESUME COMPLETE EVENT.
318
FIND OUT WHAT KIND POWER MANAGEMENT EVENT BESIDE SUSPEND/RESUME EVENT AND PROCESS ACCORDINGLY.
322
E

FIG. 6c

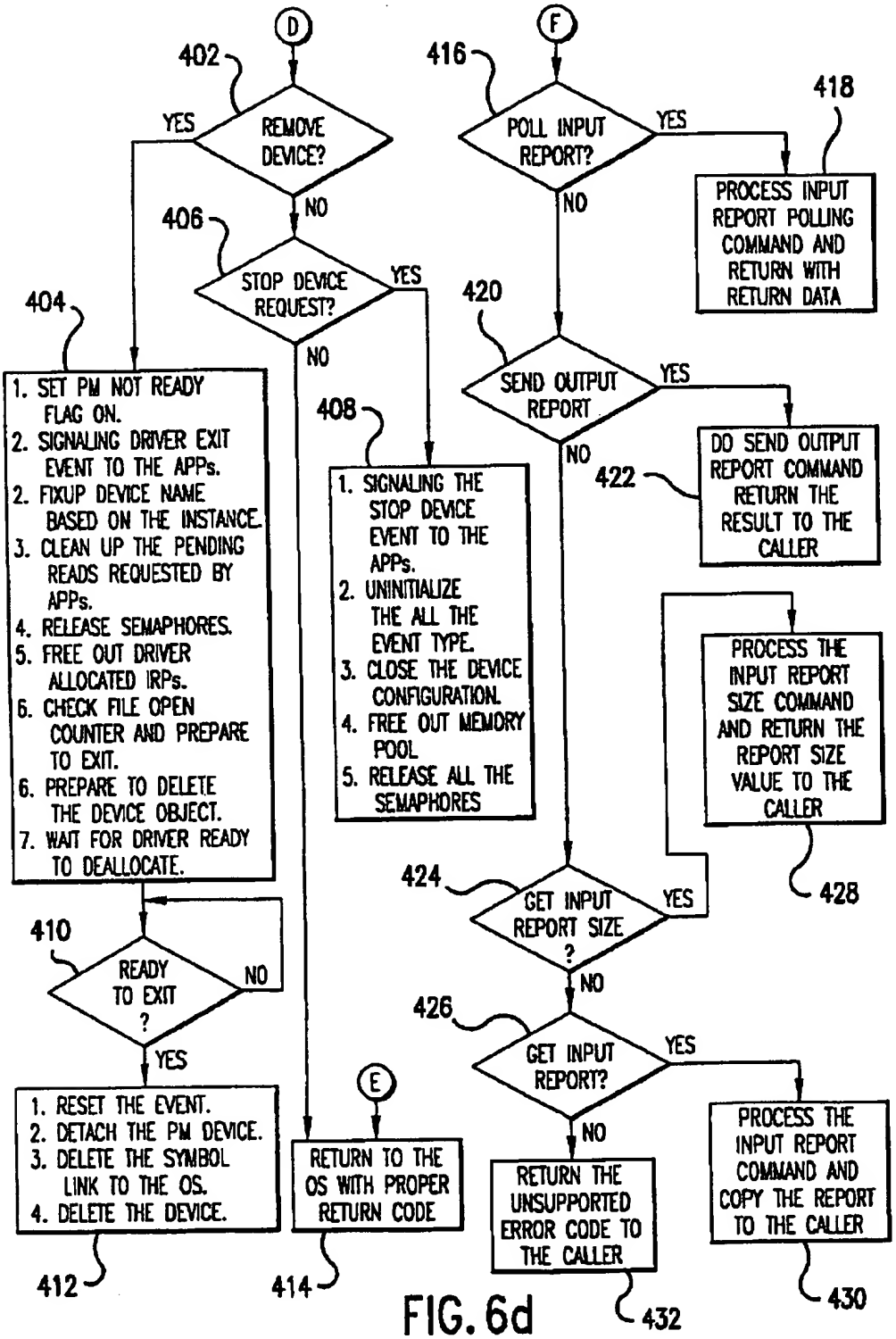

D
402
REMOVE DEVICE?
YES
NO
406
STOP DEVICE REQUEST?
YES
NO
404
1. SET PM NOT READY FLAG ON.
2. SIGNALING DRIVER EXIT EVENT TO THE APPs.
2. FIXUP DEVICE NAME BASED ON THE INSTANCE.
3. CLEAN UP THE PENDING READS REQUESTED BY APPs.
4. RELEASE SEMAPHORES.
5. FREE OUT DRIVER ALLOCATED IRPs.
6. CHECK FILE OPEN COUNTER AND PREPARE TO EXIT.
6. PREPARE TO DELETE THE DEVICE OBJECT.
7. WAIT FOR DRIVER READY TO DEALLOCATE.
408
1. SIGNALING THE STOP DEVICE EVENT TO THE APPs.
2. UNINITIALIZE THE ALL THE EVENT TYPE.
3. CLOSE THE DEVICE CONFIGURATION.
4. FREE OUT MEMORY POOL
5. RELEASE ALL THE SEMAPHORES
410
READY TO EXIT ?
NO
YES
1. RESET THE EVENT.
2. DETACH THE PM DEVICE.
3. DELETE THE SYMBOL LINK TO THE OS.
4. DELETE THE DEVICE.
412
E
RETURN TO THE OS WITH PROPER RETURN CODE
414
F
416
POLL INPUT REPORT?
YES
NO
418
PROCESS INPUT REPORT POLLING COMMAND AND RETURN WITH RETURN DATA
420
SEND OUTPUT REPORT
YES
NO
422
DO SEND OUTPUT REPORT COMMAND RETURN THE RESULT TO THE CALLER
PROCESS THE INPUT REPORT SIZE COMMAND AND RETURN THE REPORT SIZE VALUE TO THE CALLER
428
424
GET INPUT REPORT SIZE ?
YES
NO
426
GET INPUT REPORT?
YES
NO
RETURN THE UNSUPPORTED ERROR CODE TO THE CALLER
432
PROCESS THE INPUT REPORT COMMAND AND COPY THE REPORT TO THE CALLER
430

FIG. 6d

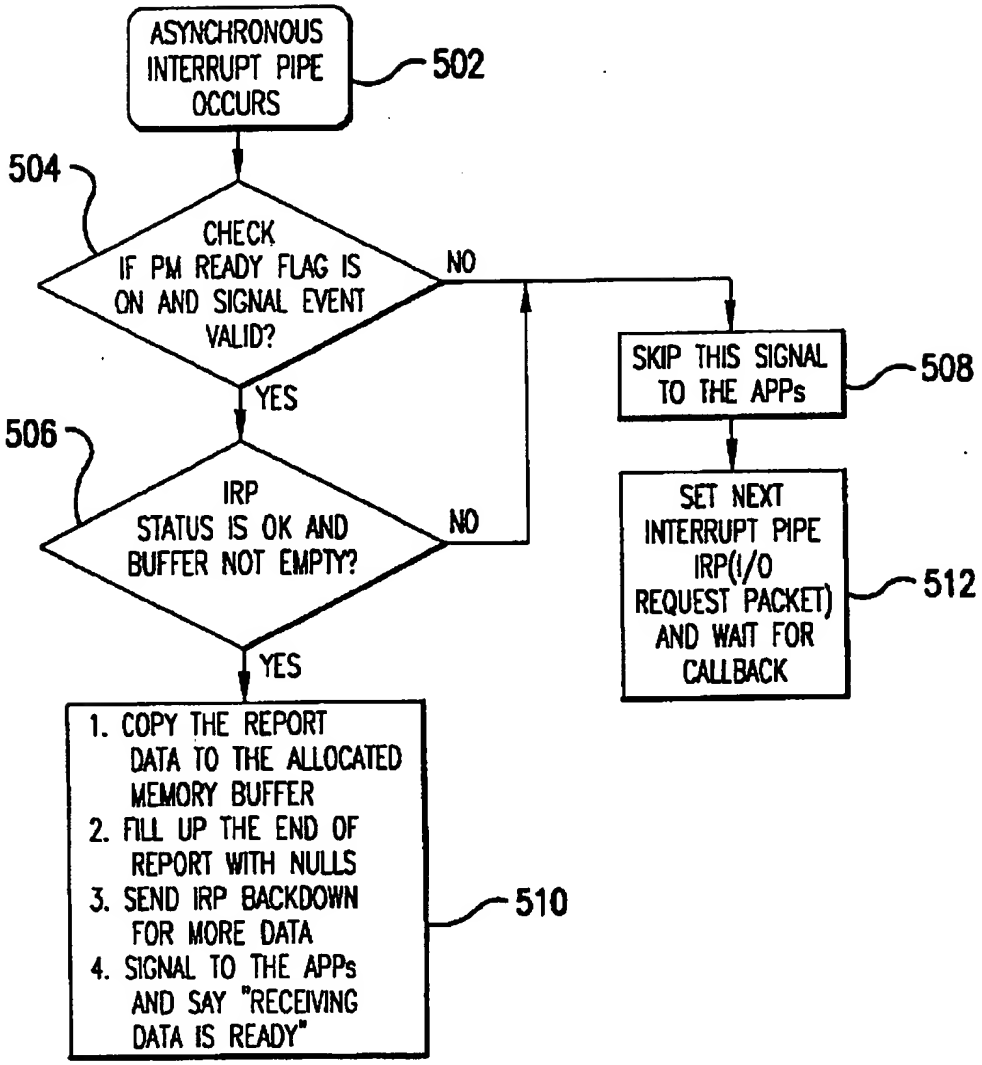

ASYNCHRONOUS INTERRUPT PIPE OCCURS
502
504
CHECK IF PM READY FLAG IS ON AND SIGNAL EVENT VALID?
NO
YES
SKIP THIS SIGNAL TO THE APPs
508
506
IRP STATUS IS OK AND BUFFER NOT EMPTY?
NO
YES
SET NEXT INTERRUPT PIPE IRP(I/O REQUEST PACKET) AND WAIT FOR CALLBACK
512
1. COPY THE REPORT DATA TO THE ALLOCATED MEMORY BUFFER
2. FILL UP THE END OF REPORT WITH NULLS
3. SEND IRP BACKDOWN FOR MORE DATA
4. SIGNAL TO THE APPs AND SAY "RECEIVING DATA IS READY"
510

FIG.7

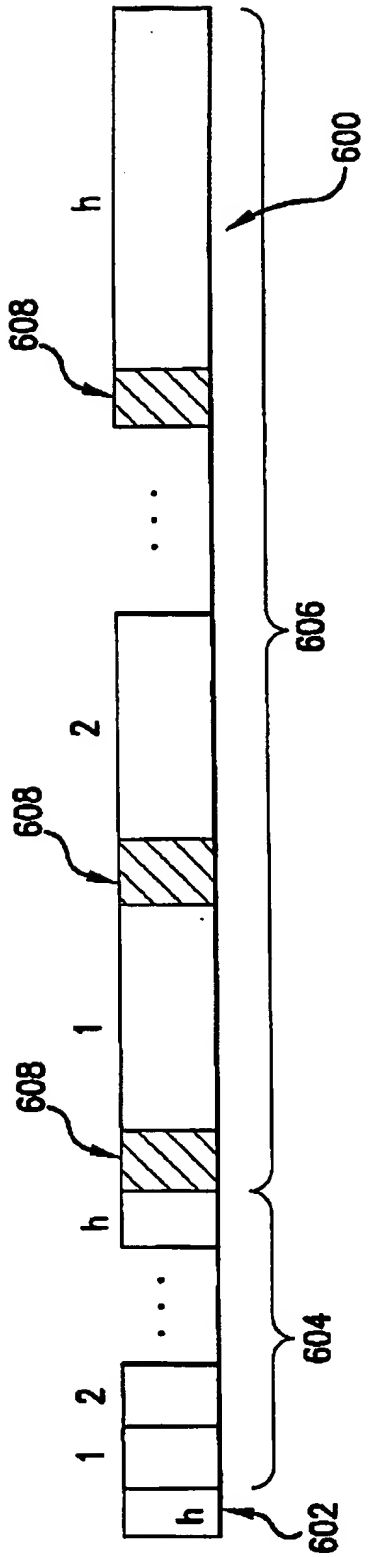


FIG.8

SYSTEM FOR HANDLING AN ASYNCHRONOUS INTERRUPT A UNIVERSAL SERIAL BUS DEVICE

APPENDIX

This application includes a Microfiche Appendix which contains 2 sheets and 121 frames, containing a source code listing and data structures. This Appendix shows an example of a particular embodiment of the present invention incorporating features thereof which are described in detail below.

BACKGROUND

1. Field of the Invention

The described embodiments are directed to peripheral devices of a computer system. In particular, the described embodiments relate to methods and processes of communication between such peripheral devices and a host unit.

2. Related Art

Personal computer (PC) designs from the early 1980s have implemented peripheral devices according to a paradigm which assigns each peripheral device a specific interrupt line. In some cases a peripheral device is assigned a direct memory access (DMA) channel. IBM and other manufactures assigned these resources to particular peripheral devices which became the standard I/O locations, interrupt lines and DMA channels used by software developers to facilitate communication with a given peripheral device.

These standard PC peripheral interfaces (e.g., serial and parallel connections) support the standard attachment of a single device. Only one peripheral device can typically be attached at any time. Thus, adding a peripheral with new design to a computer system frequently requires a costly decision of adding a new expansion card that plugs into an expansion bus (e.g., ISA/EISA or PCI) to create an attachment point for the new device. Also, these early PC designs do not support "plug and play" attachment of devices. If a new peripheral device is to be attached while the operating system of the computer system is running, the user typically needs to re-boot the computer system after attaching the device so that the Basic Input/Output System (BIOS) configures the system to associate an interrupt line with the newly attached peripheral device.

The Universal Serial Bus (USB) design standard as described in the "Universal Serial Bus Specification" (Rev. 1.0, Jan. 15, 1996) published by Compaq, Digital Equipment Corp., IBM PC Co., Intel Corp., Microsoft Corp., NEC and Northern Telecom provides a new peripheral attachment standard to overcome the above described shortcomings of the earlier scheme. In the USB architecture, multiple peripheral devices may be coupled to a single USB controller. A corresponding software driver supported by the host operating system handles inputs from the USB controller. USB systems typically include a root hub coupled to a host controller for handling all communication between the host and USB devices through a root port. USB systems support additional hubs coupled to the root port to provide additional ports for receiving USB devices.

The USB architecture categorizes peripheral devices f a computer system (e.g., a personal computer system) by classes including audio devices, communications devices, display devices, Human Interface Devices (HIDs), mass storage devices, and others. The HID class includes devices that are manipulated by humans to control some facet of a computer's perations. Devices within this class may consist of the following types:

- keyboards and printing devices;
- front panel controls;
- controls associated with devices such as telephones (dialer), VCR remote controls, and game simulation devices; and
- devices that may not control the operation of a device, but provide data in a format similar to other HID devices.

The USB architecture defines a layered software scheme which includes, at the highest level, client drivers; at an intermediate level, USB system software; and at the lowest level, host controller software. Transactions performed over the USB are controlled by the client driver. The device (or hub client) may be class specific or vendor specific. Accordingly, each hub client requires a corresponding client driver which is tailored to the specific device class or device vendor. A client driver accesses its corresponding hub client by requesting an I/O transfer using an I/O request packet (IRP). System software allocates the necessary bandwidth for the transfer and directs the IRP to its destination hub client. The host controller software communicates with a USB host controller for the actual transmission of control and data information to and from the USB devices.

The USB architecture supports four modes of transferring data between the USB devices and the host. These types of transfers are categorized as follows: Interrupt Transfers, Bulk Transfers, Isochronous Transfers, and Control Transfers. An Interrupt Transfer is used for devices that are typically thought of as interrupt driven devices. Current USB designs typically call for periodically polling the USB devices which are interrupt driven to determine whether the device has data to transfer. Control Transfers typically send specific requests to USB devices from the host. Control Transfers are typically performed during device configuration. In a Control Transfer, a special transfer sequence is used to pass requests to a device, sometimes followed by data transfer, and concluded with a status completion indication.

A main objective of the USB architecture is to allow many types of devices to be coupled as peripherals. The host typically periodically polls devices coupled to the USB to determine the characteristics of this device. In response, the polled devices provide a number of "Descriptors" to the host software to identify its characteristics. These Descriptors include a "Device Descriptor," "Configuration Descriptor," "Interface Descriptor," and "Endpoint Descriptor." HID devices typically additionally have an "Entity Descriptor" as indicated in the "Universal Serial Bus (USB) Device Class Definition for Human Interface Devices (HID) (Version 1.0, September 1996) published by the USB Implementers' Forum.

While the USB standard provides a basic architecture which may overcome the shortcomings of the earlier systems for peripheral device attachment and communication, the USB standard does not provide significant details as to how to integrate the USB architecture into a current PC design. Thus, there is a need for developing cost effective and efficient methods for implementing the USB architecture to facilitate communication with the many types of peripheral devices which may be attached to a host computer through a USB.

SUMMARY

An object of an embodiment of the present invention is to provide a system of communication between a host computer and a peripheral device associated with the applications programs executing on the host computer.

Another object of an embodiment of the present invention is to provide an implementation of a Universal Serial Bus (USB) architecture which supports communication between a USB and a single USB device which includes a plurality of controls and displays.

Another object of an embodiment of the present invention is to provide software drivers executable on a processor of a host computer which facilitate communication between a peripheral device attached to a host computer and application programs executing on the processor of the host computer.

Briefly, an embodiment of the present invention is directed to a computer system including a host having a processor and at least one port capable of attaching to at least one peripheral device. The host includes logic for associating the attached peripheral device with one of a particular peripheral device type, logic for associating the attached peripheral device with an instance of a software driver executable on the processor, and logic for enabling communication between the attached peripheral device and the host upon a detecting a condition indicating reliable data transfer between the attached peripheral device and the host. The software driver is preferably tailored to supporting communication between peripheral devices of the associated particular type and the host.

Another embodiment of the present invention is directed to a computer system including a host having a processor, and being capable of attaching to at least one peripheral device through at least one port. The host includes logic for associating an attached peripheral device with an instance of a software driver executable on the processor to support communication between the attached peripheral device and the host, and logic for associating the instance of the software driver with at least one application program executing on the processor. The instance of the software driver is preferably responsive to calls from the at least one associated application program. The host also includes logic for maintaining the instance of the software driver while the at least one associated application is executing, independent of whether the peripheral device remains attached to the host. In this manner, the applications may terminate properly, independent of the attachment of the peripheral device to the host. The host also preferably includes logic for terminating the instance of the software driver when the execution of the at least one application program terminates. This may then free the memory resources occupied by the instance.

Another embodiment of the present invention is directed to a host computer system having a USB controller capable of attaching to a USB device through a port. The USB device includes a plurality of distinct controls for receiving user inputs. The host computer system includes logic for associating the USB device with exactly one Entity Descriptor and associating each of the plurality of controls with a distinct data structure. The host computer system further includes a memory for storing a data packet received in response to an associated Interrupt Transfer initiated at the USB device. The data packet includes data representative of a plurality of the data structures. Each of the plurality of data structures preferably correspond with an input from its associated control. The host computer system further includes logic for extracting each of the data structures from the stored data packet. In this manner, inputs from multiple controls on the USB device may be transmitted to applications executing on the host in a single Interrupt Transfer.

Yet another embodiment f the present invention is directed to a host computer system capable of attaching to a peripheral device through a port to receive data transfers, each data transfers having a data packet associated therewith. The host computer system preferably includes logic for enabling the receipt of data transfers from the peripheral device in response to an attachment of the peripheral device to the host computer, and circuitry for detecting an a data transfer at the port. The host computer system further includes logic for selectively storing the data packet of the detected data transfer in an allocatable memory when at least one of the host computer system is enabled to receive data transfers and sufficient memory is allocatable to store the data packet. Thus, applications executing on the host computer system may asynchronously receive inputs from the peripheral device.

BRIEF DESCRIPTION OF THE FIGURES

FIG. 1 shows a computer system which includes a Universal Serial Bus (USB) according to an embodiment.

FIG. 2 shows a first embodiment of a control and display panel of the USB peripheral device of the embodiment of FIG. 1.

FIG. 3 shows a second embodiment of a control and display panel of the USB peripheral device of the embodiment of FIG. 1.

FIG. 4 shows an embodiment of the software layers executing on a processor of the computer system of FIG. 1 for facilitating communication with a USB peripheral device.

FIG. 5 shows a functional flow diagram of an embodiment of the device driver initialization process of the HID software driver of FIG. 4.

FIGS. *6a* through *6d* show a functional flow diagram of an embodiment of the process for handling an input/output request packet at the HID software driver of FIG. 4.

FIG. 7 shows a functional flow diagram of an embodiment of the process for handling an asynchronous interrupt transfer from the USB device at the HID software driver of FIG. 4.

FIG. 8 shows the format of a portion of a data packet associated with in an Interrupt Transfer according to an embodiment.

DETAILED DESCRIPTION

An embodiment of the present invention is directed to a Universal Serial Bus (USB) system for facilitating communication between a host unit and an enhanced Human Interface Device (HID). This system includes a protocol for transferring the information from the enhanced HID to the host computer and logic at the host computer configured to receive and transmit information to and from the enhanced HID. Other embodiments include logic for implementing plug and play capabilities. This logic may be included in a client sofiware driver for the enhanced HID, and enhancements to the system software.

According to an embodiment, the enhanced HID includes a number of different features which are preferably supported by either communication from the enhanced HID to the host computer r communication from the host computer to the enhanced HID. Such features supported by communication from the host computer to the enhanced HID may

include, for example, an answering machine light and an LED or LCD display. Such feature supported by communication from the enhanced HID to the host computer may include, for example, a dial (e.g., a volume knob), a device for providing mouse-like inputs, buttons which provide a power or sleep-mode switch, and an infrared pointer for providing VCR-like commands (e.g., play, fast forward, reverse). Thus, the enhanced HID preferably communicates with the host unit through the USB bidirectionally to support its functions.

As discussed in greater detail below, the enhanced HID is preferably associated with an "Entity Descriptor" which supports the communication from the enhanced HID to the host computer. The Entity Descriptor preferably establishes "Input Reports" which are data structures defining the format of the data transmitted from the enhanced HID to the host computer. "Output Reports" preferably define the format of the data transmitted from the host computer to the enhanced HID. Thus, the host software preferably supports the bi-directional communication according the Input and Output Reports.

Once the client driver is configured to communicate with the enhanced HID, the enhanced HID can send Input Reports for the input features (e.g., buttons, dial, and infrared pointer) for inputs to the client driver. According to an embodiment, this is accomplished through the Interrupt Transfer method as discussed above. The client driver sends data in Output Reports to the enhanced HID to support the output features (e.g., answering machine light and display). This is preferably accomplished using the Control Transfer method of transferring data from the host to the enhanced HID as discussed above. Accordingly, a single physical USB device can provide, in effect, multiple virtual input and output devices (e.g., display, IR pointer, and dial) with a single USB port and device driver.

FIG. 1 shows the computer system which includes a personal computer (PC) host **10**, a display monitor **18** which is coupled to the PC host **10** through a display connection **14**, and a Universal Serial Bus (USB) device **16** coupled to the personal computer host **10** through a USB root port **12**. The personal computer host **10** also includes USB circuitry, including a USB host controller, (not shown) as described in the "Universal Serial Bus Specification" (Rev. 1.0, Jan. 15, 1996), which is published by Compaq, Digital Equipment Corp., IBM PC Company, Intel Corp., Microsoft Corp., NEC, and Northern Telecom, and is incorporated herein by reference. Alternatively, the USB device may be coupled to the USB root port **12** through an intermediary USB hub (not shown). The personal computer host **10** also includes a microprocessor system such as that of a Pentium® design, and sufficient random access memory (RAM) to support a Windows 95® or Windows NT® operating system. The personal computer host **10** also preferably hosts a Windows 95 or Windows NT operating system. Alternatively, other operating systems which support the USB architecture as specified in the "Universal Serial Bus Specification" and the "Universal Serial Bus Device Class Definition for Human Interface Devices (HID), (Version 1.0 Draft, September 1996) which is published by the USB Implementers Forum and is incorporated herein by reference.

As shown in FIGS. **2** and **3**, the USB device **16** is preferably of an HID class and includes various controls and displays which are enabled by bidirectional communication with the personal computer host **10** through the USB hardware and software implemented in the personal computer host **10**. The USB device **16** also responds to inputs from an infrared controller (IRC) **15**.

FIG. **2** shows a control panel **20** of the USB device **16** according to a first embodiment. The user may provide inputs to the USB device **16** by pressing one of several function selection buttons **26, 27, 28, 30,** or **32** to select a particular function. Depending on the functi n selected, the user may also provide additional inputs by turning a volume knob **22**, or pressing any of buttons **38, 40, 42,** or **44**, to initiate VCR-like commands. The user may also provide the same inputs through controls on the IRC **15** (FIG. 1) which initiates signals which are detected and received at an infrared receiver (IRR) **24**. Each of the function selection buttons **26, 27, 28, 30,** and **32** are associated with a small light located directly above the respective function selection button as shown in FIG. **2**. Each of the lights, including an LED, for example, preferably indicate whether the associated function has been selected or unselected/deselected. Additionally, message light **34** preferably indicates to the user when a telephone message has been received when the personal computer host **10** including telephony hardware (not shown), has been configured to function as a telephone answering machine.

FIG. **3** shows a second embodiment of a control panel **46** of the USB device **16**. The control panel **46** differs from the control panel **20** shown in FIG. **2** in that the control panel **46** includes an LCD display **62** which replaces the LEDs associated with each of the function selection buttons **26, 27, 28, 30,** and **32**. The LCD **62**, in addition to showing the selection status of the function selection buttons, illuminates a status display with respect to the VCR-like functions selectable by the buttons **38, 40, 42,** and **44**. The LCD **62** may also provide other status information.

User inputs to the controls of control panels **20** and **46** are preferably provided to applications executing on the personal computer host **10** preferably through Interrupt Transfers to the USB host controller. Additionally, applications executing on the personal computer host **10** provide status information to be displayed on the displays (i.e., the status of message light **34** and LEDs associated with function selection buttons **26, 27, 28, 30,** and **32**, and information to be displayed on the LCD **62**) via Control Transfers through the USB host controller.

FIG. **4** shows layers of software executing on the microprocessor system of the personal computer host **10** to support the interaction between applications programs **84** and the USB device **16** through the USB. A USB driver **78** (USBD) preferably executes at a higher privilege level than applications programs **84**. A human interface device (HID) driver **83** preferably executes at a privilege higher than that of the applications programs **84** but no higher than that of the USBD **78**. According to an embodiment, USBD **78** program is provided as part of a Windows 95 or Windows NT operating system, or as a compatible addition to one of these operating systems. Accordingly, the HID driver **83** may communicate with the USB through a USB driver interface (USBDI) which has been established by Microsoft Corp. in its WDM USB Driver Interface.

The HID driver **83** includes two modules, the PMAPI.DLL module **82** and the Enhan_HID.SYS module **80**. The PMAPI.DLL module **82** in a Win32 scheme, is preferably an applications program interface (API) dynamic-link library (DLL) which exports callable functions to the applications programs **84**. Accordingly the PMAPI.DLL **82** functions preferably support Win32 based applications. Thus, to send status information to the USB device **16** in a Control Transfer, or to receive data from an interrupt transfer initiated at the USB device **16**, one of the applications programs **84** preferably makes a call to one or more functions in the PMAPI.DLL module **82**.

The Enhan_HID.SYS module **80** performs several lower level functions. The Enhan_HID.SYS module **80** processes input/output request packets (IRPs) directed to the USB device **16** from either a function in the PMAPI.DLL module **82** or the operating system. The Enhan_HID.SYS module **80** also processes Interrupt Transfers initiating at the USB device **16** through the USBD **78**. Additionally, the Enhan_HID.SYS module **80** controls the initiation and termination of processes in support of the plug and play functionality of the USB device **16**.

As indicated by the function selection buttons **26, 27, 28, 30,** and **32,** on the control panel of the USB device **16** (FIGS. **2** and **3**), the USB device **16** provides the user with a simple, convenient, and centralized control of the personal computer host **10** when functioning as a television, an FM radio, a CD/DVD player, a telephone, or a telephone answering machine. Accordingly, the applications programs **84** correspond with the applications programs, used in conjunction with relative hardware devices, which enable the personal computer host **10** to perform these selectable functions as is understood by those of ordinary skill in the art.

A USB device is generally characterized by several "Descriptors" including a "Device Descriptor," one or more "Configuration Descriptors," one or more "Interface Descriptors" associated with each Configuration Descriptor, one or more "Endpoint Descriptors" associated with each Interface Descriptor as discussed in detail in the Universal Serial Bus Specification (Revision 1.0) incorporated herein by reference above. Specific USB devices classified as Human Interface Devices (HID) may be associated with additional descriptors including a "HID Descriptor" associated with an Interface Descriptor, and one or more "Entity Descriptors" associated with the HID Descriptor a discussed in detail in the Universal Serial Bus device Class Definition for Human Interface Devices (Version 1.0) incorporated by reference above. According to an embodiment, the USB device **16** is classified as an HID USB device which is associated with one Device Descriptor, one Configuration Descriptor, one Interface Descriptor, one Endpoint Descriptor, one HID Descriptor, and one Entity Descriptor.

According to an embodiment, upon detection of an attachment of the USB device **16** to the root port **12**, or the port of an intermediary hub, the USB interrogates the USB device **16** using a Control Transfer to obtain the associated Device Descriptor, Configuration Descriptor, Interface Descriptor, Endpoint Descriptor, and HID Descriptor. The Entity Descriptor defines one or more "Input Reports" which provide inputs to the applications programs **84** in response to user inputs at the USB device **16**. Such Input Reports are preferably included in a serial data packet which is associated with an Interrupt Transfer received at the USB host controller. As discuss in greater detail below, this serial data packet is to be written to a buffer location in memory upon certain conditions to be later retrieved by the applications programs **84**, through the HID driver **83**.

As discussed above, the applications programs **84** from time to time send data to the USB device **16** to change the displays to inform the user of the status of various selected functions. Thus, a set of "Output Reports," are sent from the applications programs **84** to the USB device **16** using Control Transfers.

FIGS. **5**, **6***a* through **6***d*, and FIG. **7** show a flow diagram illustrating the functioning of an embodiment of the Enhan_HID.SYS module **80** in response to selected events. FIG. **5** illustrates how the Enhan_HID.SYS module **80** executes in response to an attachment of the USB device **16** to the USB. FIG. **7** illustrates the functioning of the Enhan_HID.SYS module **80** in response to a "call back" event at the USBD **78** indicating an Interrupt Transfer. FIG. **6***a* through **6***d* illustrates the functioning of the Enhan_HID.SYS module **80** in response to an IRP originating at the applications programs **84** or the operating system. As discussed in greater detail below, these IRPs include requests for, among other things, initializing an attached USB device **16**, establishing event handlers associated with the USB device **16** to enable call back events to associated applications, retrieving data representative of Input Reports received from the USB device **16**, and sending data representative of Output Reports to the USB device **16**.

At FIG. **5**, step **101**, the USB detects an attachment of the USB device **16** and receives information representative of the Device Descriptor associated with the USB device **16** as discussed above. The received Device Descriptor information may indicate whether the USB device **16** is of the type shown in FIG. **2** without an LCD, or having a control panel such as that shown in FIG. **3** having the LCD display **62**. Step **102** checks to determine that the Device Descriptor associated with the attached USB device **16** correspond with a USB device which is supported by the Enhan_HID.SYS module **80**. If these modules support the attached device, device driver entry points for the particular attached device are installed at step **104**. This creates an instance of the routines in Enhan_HID.SYS module **80** for processing IRPs originating at the operating system or from routines in PMAPI.DLL modules **82**, and for handling Interrupt Transfers originating at the USB device **16** directed to the applications programs **84**.

Step **104** also initiates an instance of a routine in the Enhan_HID.SYS module **80** for loading and unloading the instance of these routines. Thus, once the instance is created, and hence "loaded" to memory, a routine periodically determines whether the USB device **16** is still in its attached state and whether any applications are supported by the instance. If the USB is no longer attached and no applications are supported by the instance, this routine unloads the instance to free processing resources.

It is understood that immediately following the attachment of the USB device **16** to the USB through the root port **12**, any data received from the USB device **16** may be corrupted from noise and transients. At step **105**, a Deferred Procedure Call (DPC) timer is initiated which causes a delay for a specified duration (e.g., five seconds) to allow transients and noise at the connection of the USB device **16** to the root port **12** to settle. After this specified delay, processing proceeds to step **106** which includes setting a device ready flag in a memory location to indicate that data in data packets received from the attached USB device **16** in Interrupt Transfers at the USB is reliable, and that the USB may reliably send data to the attached USB device **16** in Control Transfers. This memory location including the device ready flag is preferably write protected at the privilege level of the Enhan_HID.SYS module **80**.

FIGS. **6***a* through **6***d* show a flow diagram illustrating the functioning of the Enhan_HID.SYS module **80** in processing an IRP initiated by either the operating system or an applications program (via the PMAPI.DLL module **82**). At step **110**, an IRP from either the operating system or the applications is received. Step **112** determines the current stack location of the IRP, and other information related to, or associated with, the attached device to which the IRP is directed using Microsoft functions provided through the USBDI.

At step **114**, the Enhan_HID.SYS module **80** determine whether there are sufficient resources to process the IRP. If

there are not sufficient resources to process the IRP, a proper error message is generated and control is returned to the OS at step 116. Step 118 checks to determine whether the IRP is a request to start the device. Either the OS or application may request the Enhan_HID.SYS module 80 to start the device upon detection of the attachment f the USB device 16 to the USB. Thus, at step 120, the Enhan_HID.SYS module 80 obtains the Configuration, Interface, and Endpoint Descriptors. An Interrupt Request Pipe from the USB device to the USB stack is initialized. Also, the DPC timer is initialized to begin a wait sequence which terminates when the PM hardware ready flag is set as discussed above in connection with FIG. 5.

Steps 122, 124, 126, and 128 illustrate a feature which monitors the number of distinct applications which are to be supported by a singular USB device 16 associated with an instance of a device driver of the Enhan_HID.SYS module 80 initiated at step 104. The Enhan_HID.SYS module 80 maintains a counter of the number of distinct applications which are supported by the instance of the device driver. Thus, to obtain support of the instance of the device driver, an application preferably sends an IRP to "open" the device which causes the counter to be incremented at step 124. When an application no longer requires support of the USB device 16, or its associated instance of the driver, the application sends an IRP to "close" the device, thereby decreasing the counter at step 128. As such, the Enhan_HID.SYS module 80 will not unload the drivers for the USB device 16 to free up resources (as discussed above in connection with step 104) until the counter is decremented to zero.

As discussed above, the USB device 16 and its associated instance of the driver may support more than one distinct application. For each of these applications, the Enhan_HID.SYS module 80 preferably maintains an instance of the software driver associated with each of the supported applications. Turning to FIG. 6b, step 206 determines whether the incoming IRP is to create an additional instance of the software driver to support an additional application. If so, step 208 performs the necessary function to determine the pipe attributes, and the interface requirements for the additional instance to be associated with the application to be supported by the additional instance, using, for example, standard Microsoft provided routines.

It is understood that certain applications need to be apprised of certain events occurring at the USB device 16 (such as buttons being pushed, knobs being turned, etc.) and other applications need not be apprised of any events occurring at the USB device 16. For those applications requiring knowledge of such events, such applications may initiate an IRP which is then detected at step 210 to request a single event handler. Upon detection of such request, step 212 determines the type of signaling required to inform the particular application of such events. For example, if the applications are hosted on an operating system which uses a Windows Driver Model (WDM) different from that of the Enhan_HID.SYS module 80, step 212 preferably converts its signaling to the applications to accommodate the WDM-type of the operating system hosting the applications.

At step 214, the Enhan_HID. SYS module 80 responds to IRPs initiated by an applications programming interface (API) function call from routines in the PMAPI.DLL module 82. These are discussed in greater detail with reference to FIG. 6c. Step 216 detects an IRP from the operating system requesting an indication as to the attachment state of the USB device 16 from the USB. Upon such a detection, step 218 determines the proper WDM format of the operating system so that the handler indicating such a detachment event to the operating system is in the proper compatible WDM format.

Step 220 detects an IRP from the operating system requesting particular Device Descriptors. For example, the operating system may be requesting information as to whether the attached USB device 16 is of the type having a control panel 20 as shown in FIG. 2 (without an LCD display 62) or of the type having a control panel 46 as shown in FIG. 3 (with an LCD display 62). Step 222 then calls a lower layer stack in the USBD 78 to retrieve the Device Descriptor.

At FIG. 6c, continuing from point C on FIG. 6b, step 302 responds to a request from the operating system inquiring as to the number of USB devices 16 which are coupled to the root port 12. It is understood that multiple USB devices 16 may be coupled to the root port 12 through an intermediary USB hub and that multiple devices may be using distinct instances of the same driver. Accordingly, such a request initiates step 304 which returns the number of devices supported by the Enhan_HID.SYS module 80 to the caller. Step 306 detects a request from the operating system to collect various device extension information at step 308. This IRP may be issued as a part of a debugging procedure.

Steps 312, 314, 316, 318, 320, and 322 are directed to power management requests. Namely, a "suspend" request detected at step 312 and a "resume" request detected at step 316 initiate similarly described processes in the "Advanced Power Management (APM) Bios Interface Specification Revision 1.2, published by Intel Corporation and Microsoft Corporation (February, 1996) incorporated herein by reference. Thus, a power suspend request detected at step 312 initiates processes set forth in step 314 including a spindown of pending IRPs, an abort of any interrupt pipe requests, a reset of the interrupt pipe, a release of all previous IRPs, and storage of the state of the USB to memory. Detection of a power resume request at step 316 initiates processes set forth in 318 including activating a rehook flag to reinitialize the stack and re-associate any pending IRP, and performing other such resume functions, such as restoring the state of the USB. Other power management events may be detected at step 320 which would initiate certain processes to be carried out in the response to such a request at step 322.

As discussed above, a branch to point B is initiated at step 214 (FIG. 6b) upon detection of an API function request from an application. Step 324 determines whether the device ready flag has been set as discussed above in connection with FIG. 5. If the device ready flag has not been set, step 326 returns a code back to the calling application to indicate that the device is not ready. Steps 328, 330, 332, and 336 relate to establishing a protocol between the calling application and the USB device 16 which is not set forth in the USB Specification. Thus, these steps may support a proprietary interface between the calling applications programs and the USB device 16. Steps 328 and 330 relate to an initialization of that protocol. Thus, this IRP is preferably the first IRP to be called after the IRP which inquires as to whether the device ready flag is set. Conversely, an IRP requesting the processes at 336 to uninitialize the application with the USB device 16 is preferably sent prior to decreasing the device counter at step 128 (FIG. 6a). Step 334 detects a command for obtaining device information at step 338 such as, for example, information indicating as to whether the USB device 16 is of the type shown in FIG. 2 having a control panel 20 with no LCD display, or of the type having a control panel such as that shown in FIG. 3 with an LCD display 62.

It is understood that the USBD 78 may provide a signal to the operating system indicating a detachment of the USB

device 16 in response to polling by the operating system. Turning to FIG. 6d, at point D, steps 402, 404, 410 and 412 support requests from the operating system indicating a removal or detachment of the USB device 16. At step 404, the Enhan_HID.SYS module 80 resets the device ready flag which was set when the USB device 16 was attached as discussed above with reference to FIG. 5, signals to the applications associated with instances of the software driver for the USB device 16, checks the "open" counter (maintained as described above with reference to steps 122, 124, 126 and 128) to determine whether the instance of the driver may be unloaded from the system, frees interrupt pipes allocated to the instance of the software driver, releases semaphores associated with the instance of the software driver, terminates pending read requests from the applications programs 84, and prepares to delete the object associated with the USB device 16.

Step 410 illustrates a wait sequence which causes the instance of the software driver associated with the detached USB device 16 to terminate when certain conditions are met. The Enhan_HID.SYS module 80 loops back on iterations of this portion of the Enhan_HID.SYS module 80 to step 410 until the "open" counter is decremented to zero, indicating that all of the applications associated with the instance of the software driver have acknowledge receipt of a notification the detachment of the USB device 16. When the termination condition is met at step 410, step 412 performs various functions to terminate the instance of the software driver associated with the detached device by deleting the symbolic link between the operating system and the USB device 16 and deleting the instance of the software driver.

Step 406 indicates a detection of an IRP from the operating system which is a request to stop the device. This may be initiated by events which include power down, an unplugging of the device, or a user initiated "stop device" event. At step 408, the Enhan_HID.SYS modules 80 signals to the affected applications, uninitializes events associated with the device, closes the device, releases semaphores associated with the device and frees the portion of memory occupied by the instance of the software driver associated with the device.

At point F, steps 416 and 418 support requests from an application to poll the USB device using a Control Transfer. This capability may enable an application to support diagnostic functions. Steps 420 and 422 respond to a request from an application to send an Output Report to the USB device 16 using a Control Transfer as discussed above. Here, referring to the embodiments shown in FIGS. 2 and 3, such Output Reports may include information representative of changes in the status of the controls selectable from selection buttons 26, 27, 20 28, 30, 32, and the message light 34. In the embodiment shown in FIG. 2, these Output Reports may provide inputs to the USB device for changing the status of the LEDs above, and associated with, the selection buttons. In the embodiment shown in FIG. 3, these Output Reports would be used to change the state of the LCD 62.

Steps 424, 426, 428, and 430 correspond to the retrieval of Input Reports. Such Input Reports are preferably part of a data packet which is associated with an Interrupt Transfer which has been written to a buffer location in memory as discussed in greater detail with reference to FIG. 7 below. Embodiments of the control panel of the USB device 16 as shown in FIGS. 2 and 3 include function selection buttons, VCR-like controls, the volume knob 22, and the IRR 24 to receive inputs from the IRC 15. Each f these actuation features on the control panel may correspond with a separate Input Report. Thus, according to an embodiment, multiple user inputs fr m the USB device 16 may be included in a data packet from a single Interrupt Transfer.

FIG. 8 shows a portion of the data packet associated with an Interrupt Transfer which includes an n-number of Input Reports. A double word 602 at the leading portion indicates the number of input reports in the interrupt transfer data packet. N-double words follow in data field 604 which identifies the usage or function associated with each of the corresponding Input Reports which follow. Field 606 then includes these n-Input Reports. As discussed in greater detail below with reference to FIG. 7, the receipt of an Interrupt Transfer initiates a call back event directed to the applications via an associated event handler to indicate that an Interrupt Transfer has occurred and that a corresponding data packet has been stored in a buffer portion of memory. In response, the affected applications program preferably calls routines in PMAPI.DLL module 82 to send an IRP to the Enhan_HID.SYS module 80 to retrieve the number of reports in the data packet stored at the double word field 602 (FIG. 8). Then, a routine in the PMAPI.DLL modules 82 initiates another IRP to retrieve data in field 604 indicating the usage or function associated with each of the n-Input Reports in the data packet 600. The PMAPI.DLL module 82 may also initiate other IRPs to determine other structural information representative of the format of the Input Reports in the data packet.

Returning to FIG. 6d, the applications in applications programs 84 corresponding to the functions or usages identified in the data from portion 604 of the received data packet 600 sequentially initiate IRPs to first get the input report size at steps 424 and 428, and then get the report data at steps 426 and 430. Thus, as illustrated in FIG. 8, the field 608 corresponding to the associated Input Report is retrieved in response to a first IRP from the application corresponding to the usage or function of the associated Input Report. After determining the size of the Input Report from the associated double word field 608, the application initiates a second IRP to retrieve the remainder of the report at step 430. Driver run-time process reaches step 432 if the IRP does not correspond to any supported command. As such, a proper error code is generated.

FIG. 7 is a flow diagram illustrating the processing of the Enhan_HID.SYS module 80 in response to the receipt of an Interrupt Transfer. At step 502, the Enhan_HID.SYS module 80 receives a call back from the USBD 78 indicating the receipt of an Interrupt Transfer. At step 504, the process checks to see if the device ready flag is set as discussed above in connection with FIG. 5. At step 506, the process determines whether there are sufficient memory resources to receive the data packet associated with the Interrupt Transfer. If the device ready flag is set and there are sufficient resources to receive the data packet associated with the Interrupt Transfer, the process at step 510 copies the portion of the data packet including the Input Reports to an allocated buffer location, fills the end of the report portion with zeros to fill the remainder of the allocated portion, and signals to the applications that an Interrupt Transfer has been received and that an associated data packet including Input Reports has been stored in the memory buffer. In response to this signal to the applications, as discussed above, the applications may initiate IRPs to obtain the number of Input Reports in the stored data packet, the usages/functions associated with each of the Input Reports, and other format information. If step 504 determines that the device ready flag is not set, or if step 506 determines that there are not sufficient resources to receive the data packet associated with the Interrupt Transfer, step 508 bypasses the signal to the

applications and step 512 sets the next interrupt IRP and puts the Enhan_HID.SYS 80 in a state of waiting for another call back event.

While the description above refers to particular embodiments of the present invention, it will be understood that many modifications may be made without departing from the spirit thereof. The accompanying claims are intended to cover such modifications as would fall within the true scope and spirit of the present invention.

The presently disclosed embodiments are therefore to be considered in all respects as illustrative and not restrictive, the scope of the invention being indicated by the appended claims, rather than the foregoing description, and all changes which come within the meaning and range of equivalency of the claims are therefore intended to be embraced therein.

What is claimed is:

1. In a computer system including a host having a processor, a Universal Serial Bus (USB) host controller and a port for attaching to a USB device, the USB host controller being capable of coupling to the USB device through an attachment to the port, the USB host controller including circuitry for detecting an attachment of the USB device to the port for enabling a bidirectional communication channel between the host unit and the USB device, the improvement comprising:

logic for detecting a condition indicative of a degree of reliability of the bi-directional communication channel upon attachment of USB device to the at least one port; and

logic for enabling at least one of transmission of a Control Transfer from the host to the attached USB device and receipt of data provided by the attached USB device to the host in an Interrupt Transfer upon detecting the condition indicative of a degree of reliability in the bi-directional communication channel enabled by the attachment of the USB device.

2. The computer system of claim 1, wherein the enabling logic enables the at least one of transmission of a Control Transfer from the host to the attached USB device and receipt of data provided by the attached USB device to the host in an Interrupt Transfer following a set time period following a detection of an attachment of the USB device to the USB controller.

3. The computer system of claim 1, wherein the enabling logic inhibits the transmission of a Control Transfer from the host to the attached USB device and the receipt of data provided by the attached USB device to the host in an Interrupt Transfer until the detection of the condition indicative of a degree of reliability in a bi-directional communication channel enabled by the attachment of the USB device.

4. In a computer system including a host having a processor, a Universal Serial Bus (USB) host controller and a port for attaching to a USB device, the processor executing at least one applications program and one or more instances of a software driver, the software driver being capable of supporting communication between an attached USB device and the host, the USB host controller being capable of coupling to the USB device through an attachment to the port, the USB host controller including circuitry for detecting an attachment of the USB device to the port or a detachment of the USB device from the port, the improvement comprising:

logic for associating an attachment of the USB device with an instance of the software driver;

logic for associating the instance of the software driver with at least one application program executing on the processor, the instance of the software driver being responsive to calls from the at least one associated application program;

logic for maintaining the instance of the software driver while the at least one application is executing, independent of whether the USB device remains attached to the port; and

logic for terminating the instance of the software driver in response to one of a termination of the execution of the at least one application program and an acknowledgment that the application program has been notified of a detachment of the USB device.

5. The computer system of claim 4, the improvement further including:

logic for maintaining a count of the executing application programs which are associated with the instance of the software driver and decrementing the count when one of the execution of an associated application terminates and an acknowledgment that the associated application has been notified of a detachment of the USB device; and

logic for terminating the instance of the software driver when the count is decremented to zero.

6. The computer system of claim 4, the host further including a portion of memory which is allocated to the instance of the software driver, wherein the improvement further includes logic for de-allocating the memory allocated to the instance of the software driver when the instance of the software driver terminates.

7. A host computer system having a Universal Serial Bus (USB) host controller capable of attaching to a USB device through a port, the USB device having a plurality of distinct controls for receiving user inputs, the host computer system comprising:

logic for associating the USB device with exactly one Entity Descriptor which defines a plurality of data structures, each data structure being associated with one of the distinct controls;

a memory for storing a data packet received in response to an associated Interrupt Transfer initiated at the USB device, the data packet including data representative of a plurality of said data structures, each of the plurality of data structures including data representative of user inputs from its associated control; and

logic for extracting each of the plurality of data structures from the stored data packet.

8. The host computer system of claim 7, the host computer system further including:

a processor for executing application programs thereon;

logic for associating each of the reports extracted from the stored data packet with an application program; and

logic for notifying each application program associated with an extracted data structure of the receipt of said extracted data structure.

9. The host computer system of claim 7, wherein the logic for extracting data structures from the stored data packet includes:

logic for determine the number of data structures included in the stored data packet;

logic for associating each of the data structures in the stored data packet with one of the controls of the USB device;

logic for determining a memory location of each data structure in the stored data packet; and

logic for retrieving each data structure in the stored data packet at its associated memory location by executing an application program associated with the data structure.

10. A host computer system having a Universal Serial Bus (USB) controller capable of attaching to a USB device through a port to receive Interrupt Transfers, the Interrupt Transfers having a data packet associated therewith, the host computer system comprising:

logic for enabling the receipt of Interrupt Transfers from the USB device at the USB controller in response to an attachment of the USB device to the USB controller;

circuitry for detecting an Interrupt Transfer at the port; and

logic for selectively storing the data packet of the detected interrupt transfer in an allocatable memory when at least one of the host computer system is enabled to receive Interrupt Transfers and sufficient memory at the host computer system is allocatable to store the data packet.

11. The host computer system of claim 10, wherein the logic for enabling the receipt of Interrupt Transfers further includes:

logic for associating the attached USB device with an instance of a software driver;

logic for detecting of a condition indicative of a degree of reliability of data received from Interrupt Transfers initiated at the attached USB device.

12. The host computer system of claim 11, wherein the logic detecting a degree of reliability of the data received from Interrupt Transfers initiated at the attached USB device detects a passage of a set time period following a detection of an attachment of the USB device to the USB controller.

13. The host computer system of claim 10, the host computer system further comprising logic for transmitting an error signal to the attached USB device upon detection of an Interrupt Transfer when the host computer system is not enabled to receive the Interrupt Transfer or there is not sufficient memory allocatable to store the associated data packet.

14. A computer readable medium for use in conjunction with a computer system including a host having a processor and a Universal Serial Bus (USB) host controller, the USB host controller being capable of attaching to at least one USB device through at least one port, the computer readable medium including computer readable instructions encoded thereon for:

detecting an attachment of a USB device to the at least one port; and

enabling at least one of transmission of a Control Transfer from the host to the attached USB device and receipt of data provided by the attached USB device to the host in an Interrupt Transfer upon detection of a condition indicative of a degree of reliability in a bi-directional communication channel enabled by the attachment of the USB device.

15. The computer readable medium of claim 14, the computer readable medium further including computer readable instructions encoded thereon for enabling the at least one of transmission of a Control Transfer from the host to the attached USB device and receipt of data provided by the attached USB device to the host in an Interrupt Transfer following a set time period following a detection of an attachment of the USB device to the USB controller.

16. The computer readable medium of claim 14, the computer readable medium further including computer readable instructions encoded thereon for inhibiting the transmission of a Control Transfer from the host to the attached USB device and the receipt of data provided by the attached USB device to the host in an Interrupt Transfer until the detection of the condition indicative of the degree of reliability.

17. A computer readable medium for use in conjunction with a computer system including a host having a processor and a Universal Serial Bus (USB) host controller, the USB host controller being capable of attaching to at least one USB device through at least one port, the computer readable medium including computer readable instructions encoded thereon for:

associating an attached USB device with an instance of a software driver executable on the processor, the software driver being capable of supporting communication between the attached USB device and the host;

associating the instance of the software driver with at least one application program executing on the processor, the instance of the software driver being responsive to calls from the at least one associated application program;

maintaining the instance of the software driver while the at least one application is executing, independent of whether the USB device remains attached to the USB; and

terminating the instance of the software driver in response to one of a termination of the execution of the at least one application program and an acknowledgment that the application program has been notified of a detachment of the USB device.

18. The computer readable medium of claim 17, the computer readable medium further including computer readable instructions encoded thereon for:

maintaining a count of the executing application programs which are associated with the instance of the software driver and decrementing the count when one of the execution of an associated application terminates and an acknowledgment that the associated application has been notified of a detachment of the USB device; and

terminating the instance of the software driver when the count is decremented to zero.

19. The computer readable medium of claim 17, wherein the host further includes a portion of memory which is allocated to the instance of the software driver, wherein the computer readable medium further includes computer readable instructions encoded thereon for de-allocating the memory allocated to the instance of the software driver when the instance of the software driver terminates.

20. A computer readable medium for use in conjunction with a host computer system having a Universal Serial Bus (USB) host controller capable of attaching to a USB device through a port, the USB device having a plurality of distinct controls for receiving user inputs, the computer readable medium including computer readable instructions encoded thereon for:

associating the USB device with exactly one Entity Descriptor which defines a plurality of data structures, each data structure being associated with one of the distinct controls;

storing in a memory associated with the host computer system a data packet received in response to an associated Interrupt Transfer initiated at the USB device, the data packet including data representative of a plurality of said data structures, each of the plurality of data structures including data representative of user inputs from its associated control; and

extracting each of the plurality of data structures from the stored data packet.

21. The computer readable medium of claim 20, wherein the host computer system further includes a processor for executing application programs thereon, and wherein the

computer readable medium further includes computer readable instructions encoded thereon for: associating each of the reports extracted from the stored data packet with an application program; and

notifying each application program associated with an extracted data structure f the receipt of said extracted data structure.

22. The computer readable medium of claim 20, the computer readable medium further including computer readable instructions encoded thereon for:

determining a number of data structures included in the stored data packet;

associating each of the data structures in the stored data packet with one of the controls of the USB device;

determining a memory location of each data structure in the stored data packet; and

retrieving each data structure in the stored data packet at its associated memory location by executing an application program associated with the data structure.

23. A computer readable medium for use in conjunction with a host computer system having a Universal Serial Bus (USB) controller capable of attaching to a USB device through a port to receive Interrupt Transfers, the Interrupt Transfers having a data packet associated therewith, the computer readable medium including computer readable instructions encoded thereon for:

enabling the receipt of Interrupt Transfers from the USB device at the USB controller in response to an attachment of the USB device to the USB controller;

detecting an Interrupt Transfer at the port; and

selectively storing the data packet of the detected interrupt transfer in an allocatable memory when at least one of the host computer system is enabled to receive Interrupt Transfers and sufficient memory at the host computer system is allocatable to store the data packet.

24. The computer readable medium of claim 23, the computer readable medium further including computer readable instructions encoded thereon for:

associating the attached USB device with an instance of a software driver; and

detecting a condition indicative of a degree of reliability of data received from Interrupt Transfers initiated at the attached USB device.

25. The computer readable medium of claim 24, the computer readable medium further including computer readable instructions encoded thereon for detecting a passage of a set time period following a detection of an attachment of the USB device to the USB controller to determine a degree of reliability of the data received from Interrupt Transfers initiated at the attached USB device detects.

26. The computer readable medium of claim 23, the computer readable medium further including computer readable instructions encoded thereon for transmitting an error signal to the attached USB device upon detection of an Interrupt Transfer when the host computer system is not enabled to receive the Interrupt Transfer or there is not sufficient memory allocatable to store the associated data packet.

* * * * *

US006218969B1

(12) **United States Patent**
Watson et al.

(10) **Patent No.: US 6,218,969 B1**
(45) **Date of Patent: Apr. 17, 2001**

(54) **UNIVERSAL SERIAL BUS TO PARALLEL BUS SIGNAL CONVERTER AND METHOD OF CONVERSION**

(75) Inventors: **Lynn R. Watson; James E. Castleberry; David D. Luke; David C. Gilbert**, all of Boise, ID (US)

(73) Assignee: **In-System Design, Inc.**, Boise, ID (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

This patent is subject to a terminal disclaimer.

(21) Appl. No.: **09/372,344**

(22) Filed: **Aug. 11, 1999**

Related U.S. Application Data

(63) Continuation of application No. 08/974,736, filed on Nov. 19, 1997.

(51) **Int. Cl.**[7] **H03M 9/00**
(52) **U.S. Cl.** **341/100**; 710/106
(58) **Field of Search** 341/100, 101; 370/366; 395/891; 710/105, 106, 46

(56) **References Cited**

U.S. PATENT DOCUMENTS

H696	* 10/1989	Davidson	710/46
3,643,135	2/1972	Devore et al. .	
3,790,858	2/1974	Brancaleone et al. .	
3,885,167	5/1975	Berglund .	
4,027,941	6/1977	Narozny .	
4,124,888	11/1978	Washburn .	
4,217,624	8/1980	Tuck .	
4,262,981	4/1981	Goodman .	
4,295,181	10/1981	Chang et al. .	
4,342,069	7/1982	Link .	
4,361,955	12/1982	Lancaster .	
4,458,967	7/1984	King et al. .	
4,487,464	12/1984	Kirschenbaum .	
4,543,450	9/1985	Brandt .	
4,603,320	7/1986	Farago .	
4,686,506	8/1987	Farago .	
4,691,350	9/1987	Kleijne et al. .	
4,691,355	9/1987	Wirstrom et al. .	
4,694,492	9/1987	Wirstrom et al. .	
4,972,470	11/1990	Farago .	
5,663,721	9/1997	Rossi .	
5,729,573	3/1998	Bailey et al. .	
5,838,926	11/1998	Yamagishi .	
5,970,220	* 10/1999	Bolash et al.	358/1.1
6,040,792	* 3/2000	Watson et al.	341/100

OTHER PUBLICATIONS

Richard Nass, "The Universal Serial Bus . . . ", Electronic Design, all pages, Aug. 1997.*
Mark Richman, "A PC Peripheral Revolution Arrives," Computer Technology Review, all pages, Apr. 1997.*
*Universal US3 Serial Bus, Universal Serial Bus Specification Version 1.0, Jan. 19, 1996.

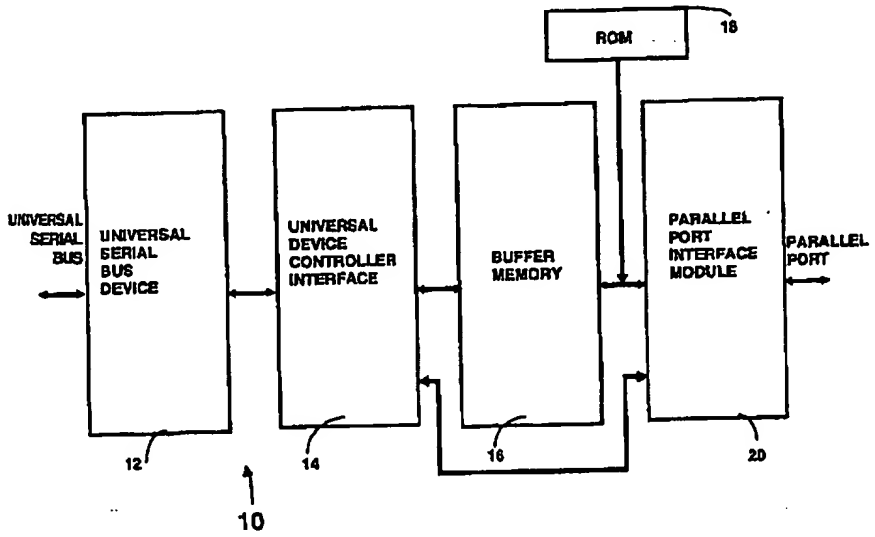
* cited by examiner

Primary Examiner—Patrick Wamsley
(74) *Attorney, Agent, or Firm*—Marger Johnson & McCollom, P.C.

(57) **ABSTRACT**

A serial to parallel port signal converter for interconnection between a hosts utilizing Uniform Serial Bus communications protocols and a peripheral device uses IEEE 1284 complaint communications protocol. The signal converter appears to the host as a fully compliant bi-directional USB device, and to the peripheral device as a fully compliant IEEE 1284 host.

8 Claims, 18 Drawing Sheets



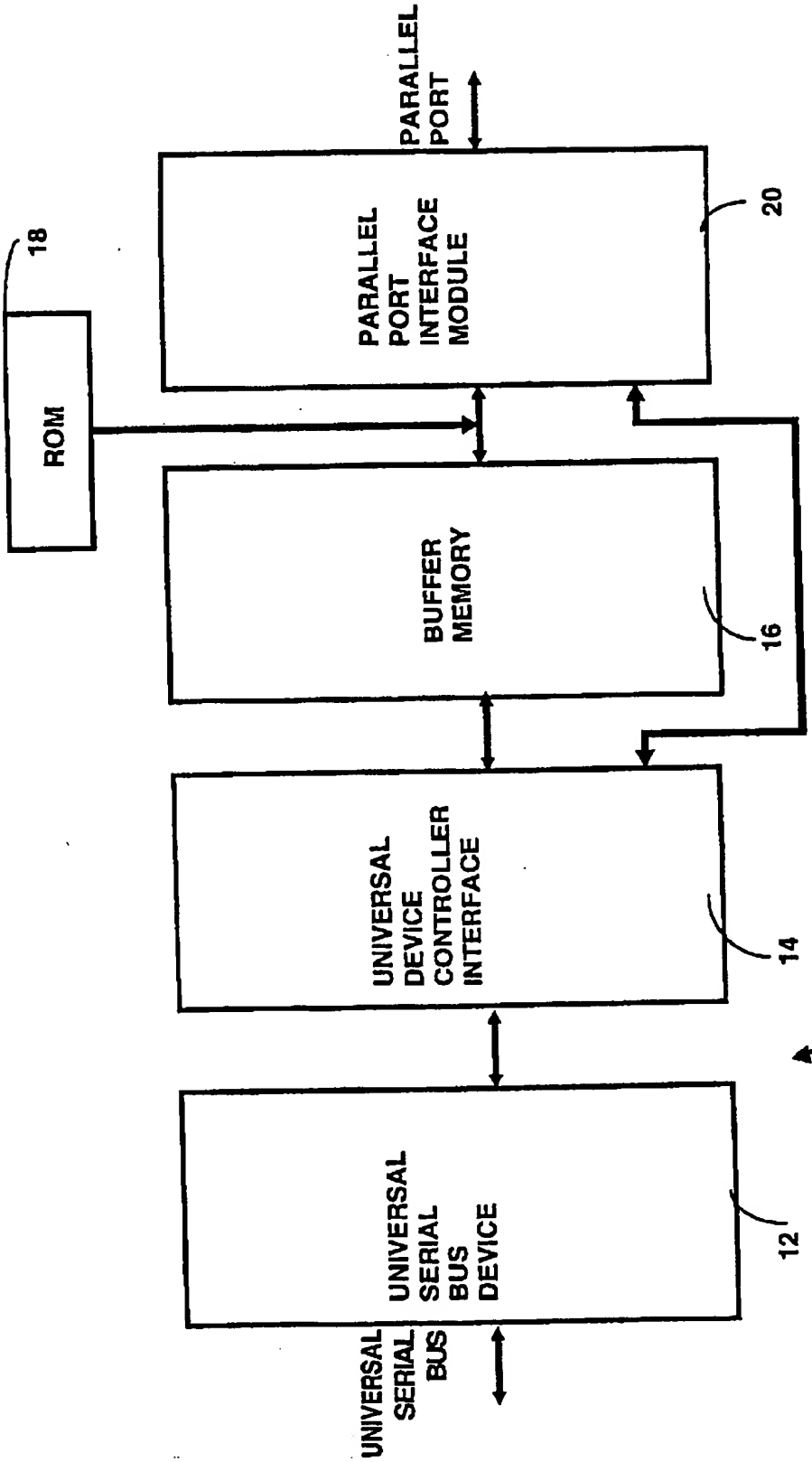

PARALLEL PORT
PARALLEL PORT INTERFACE MODULE
20
18
ROM
BUFFER MEMORY
16
UNIVERSAL DEVICE CONTROLLER INTERFACE
14
10
UNIVERSAL SERIAL BUS DEVICE
12
UNIVERSAL SERIAL BUS

FIG.1

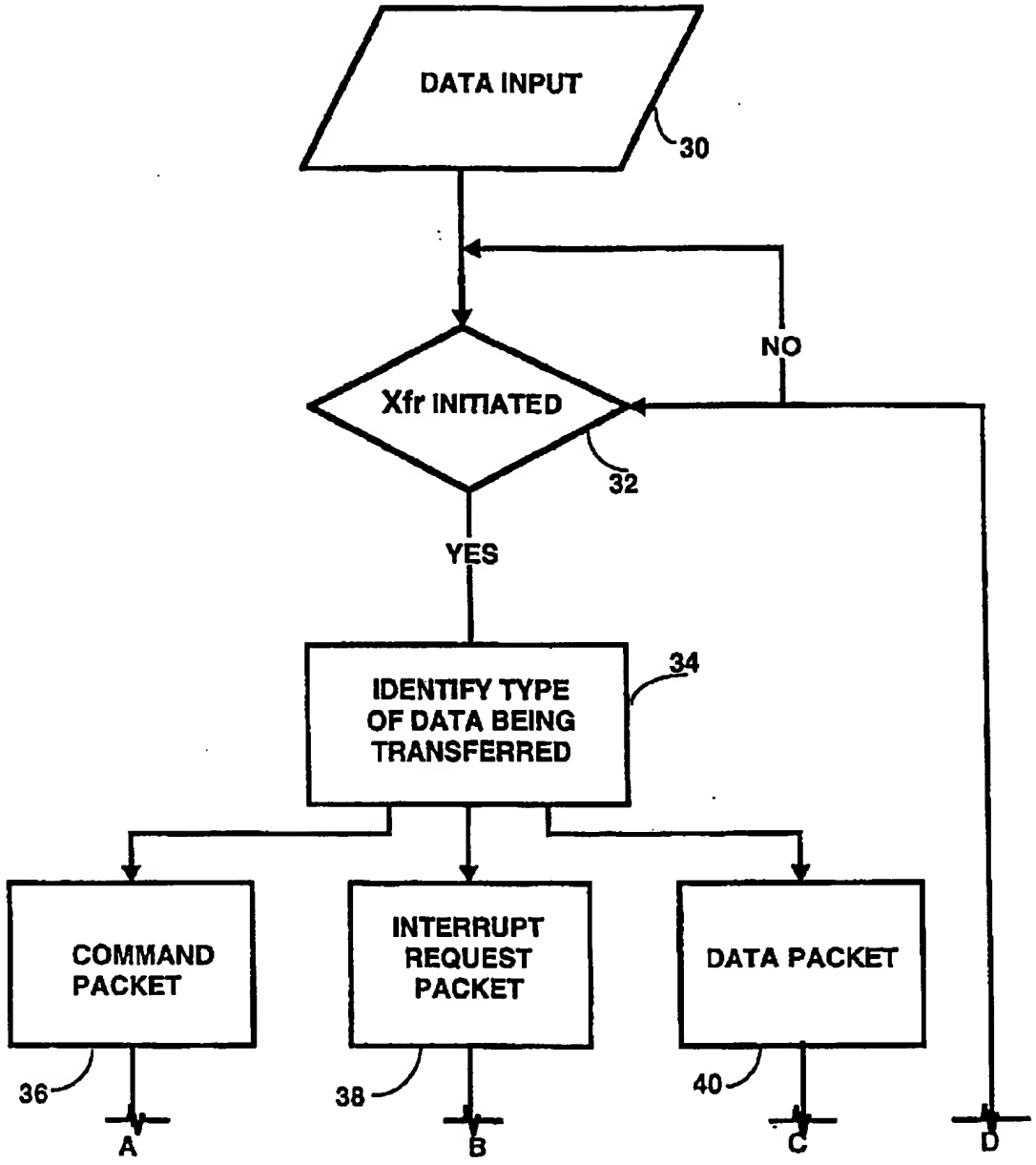

DATA INPUT
30
NO
Xfr INITIATED
32
YES
IDENTIFY TYPE OF DATA BEING TRANSFERRED
34
COMMAND PACKET
INTERRUPT REQUEST PACKET
DATA PACKET
36
38
40
A
B
C
D

FIG. 2 A

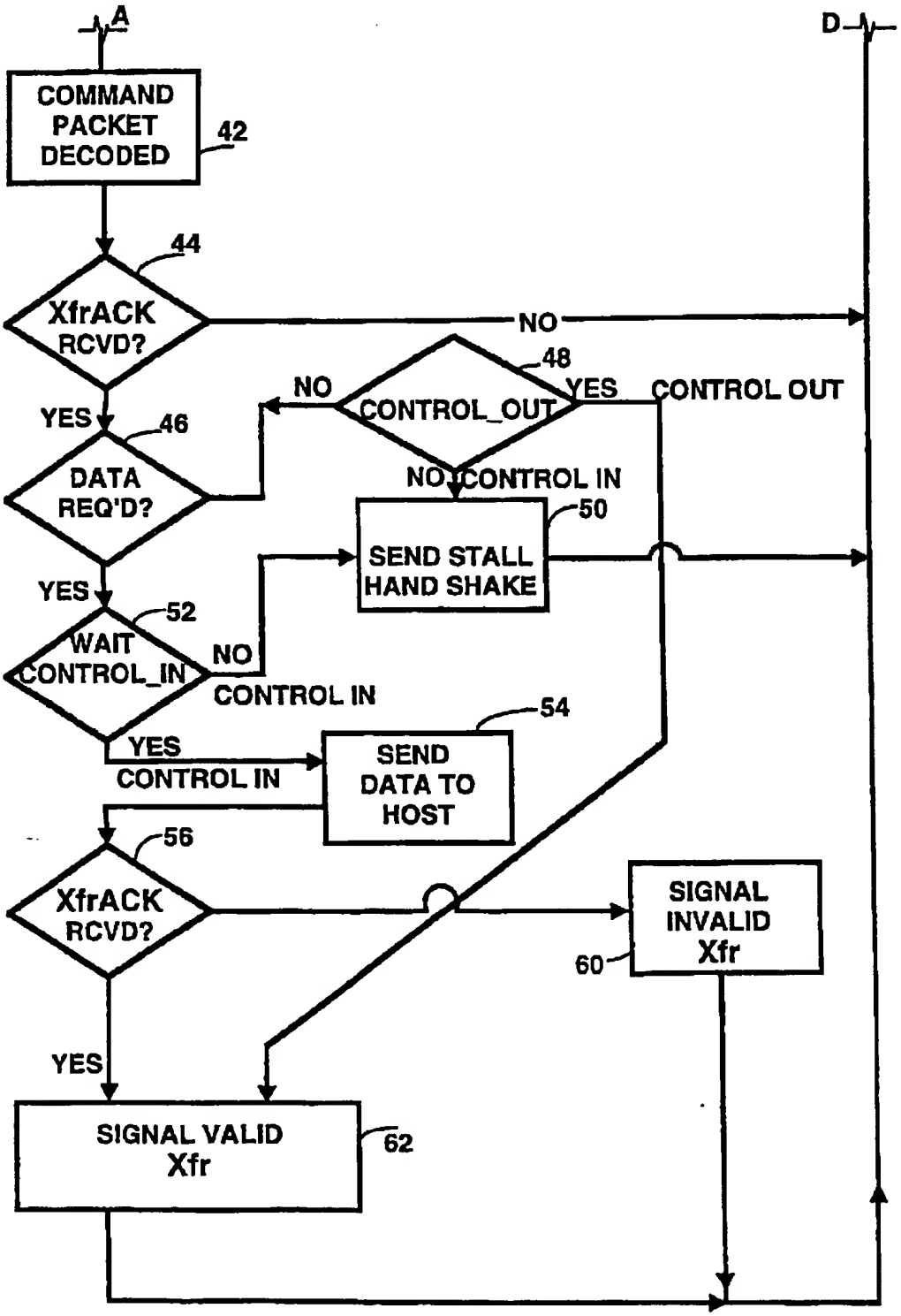

A
D
COMMAND PACKET DECODED
42
44
XfrACK RCVD?
NO
48
NO
CONTROL_OUT
YES
CONTROL OUT
YES
46
DATA REQ'D?
NO CONTROL IN
50
SEND STALL HAND SHAKE
YES
52
WAIT CONTROL_IN
NO
CONTROL IN
54
YES
CONTROL IN
SEND DATA TO HOST
56
XfrACK RCVD?
SIGNAL INVALID Xfr
60
YES
SIGNAL VALID Xfr
62

FIG. 2B

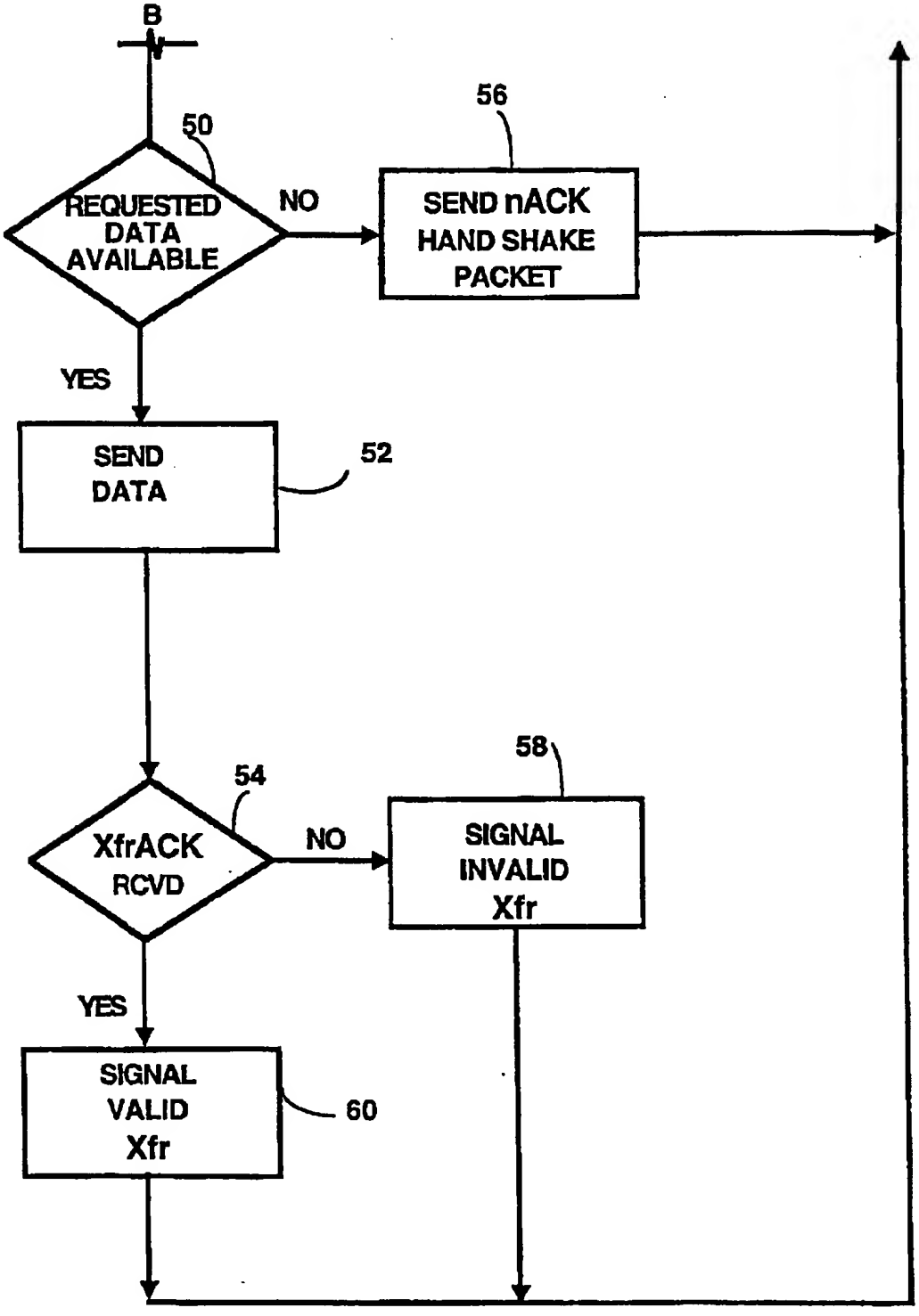

B
50
REQUESTED DATA AVAILABLE
NO
56
SEND nACK HAND SHAKE PACKET
YES
SEND DATA
52
54
XfrACK RCVD
NO
58
SIGNAL INVALID Xfr
YES
SIGNAL VALID Xfr
60

FIG. 2C

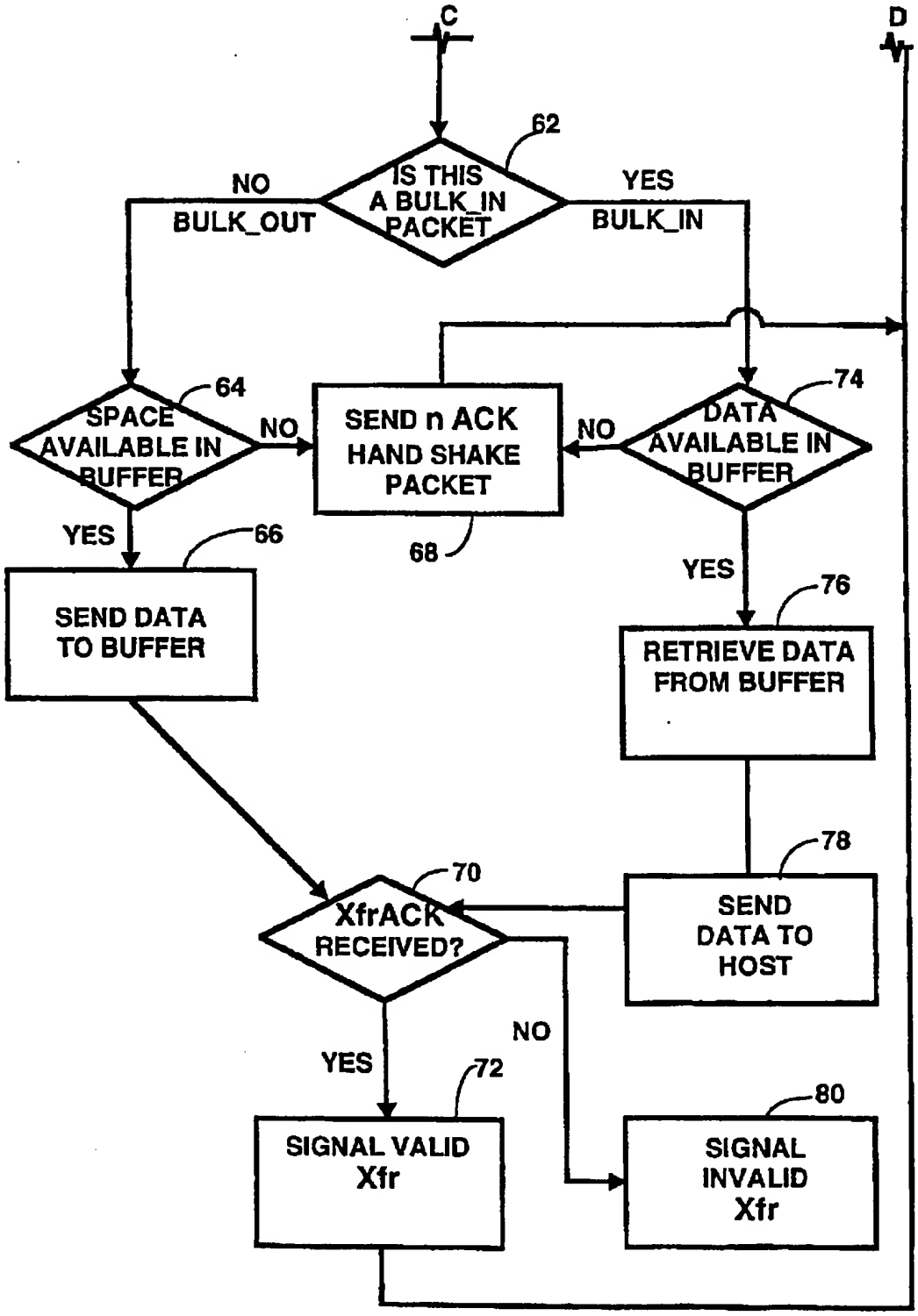

C
D
62
IS THIS A BULK_IN PACKET
NO
BULK_OUT
YES
BULK_IN
64
SPACE AVAILABLE IN BUFFER
NO
SEND n ACK HAND SHAKE PACKET
68
NO
74
DATA AVAILABLE IN BUFFER
YES
66
SEND DATA TO BUFFER
YES
76
RETRIEVE DATA FROM BUFFER
78
SEND DATA TO HOST
70
XfrACK RECEIVED?
NO
YES
72
SIGNAL VALID Xfr
80
SIGNAL INVALID Xfr

FIG. 2D

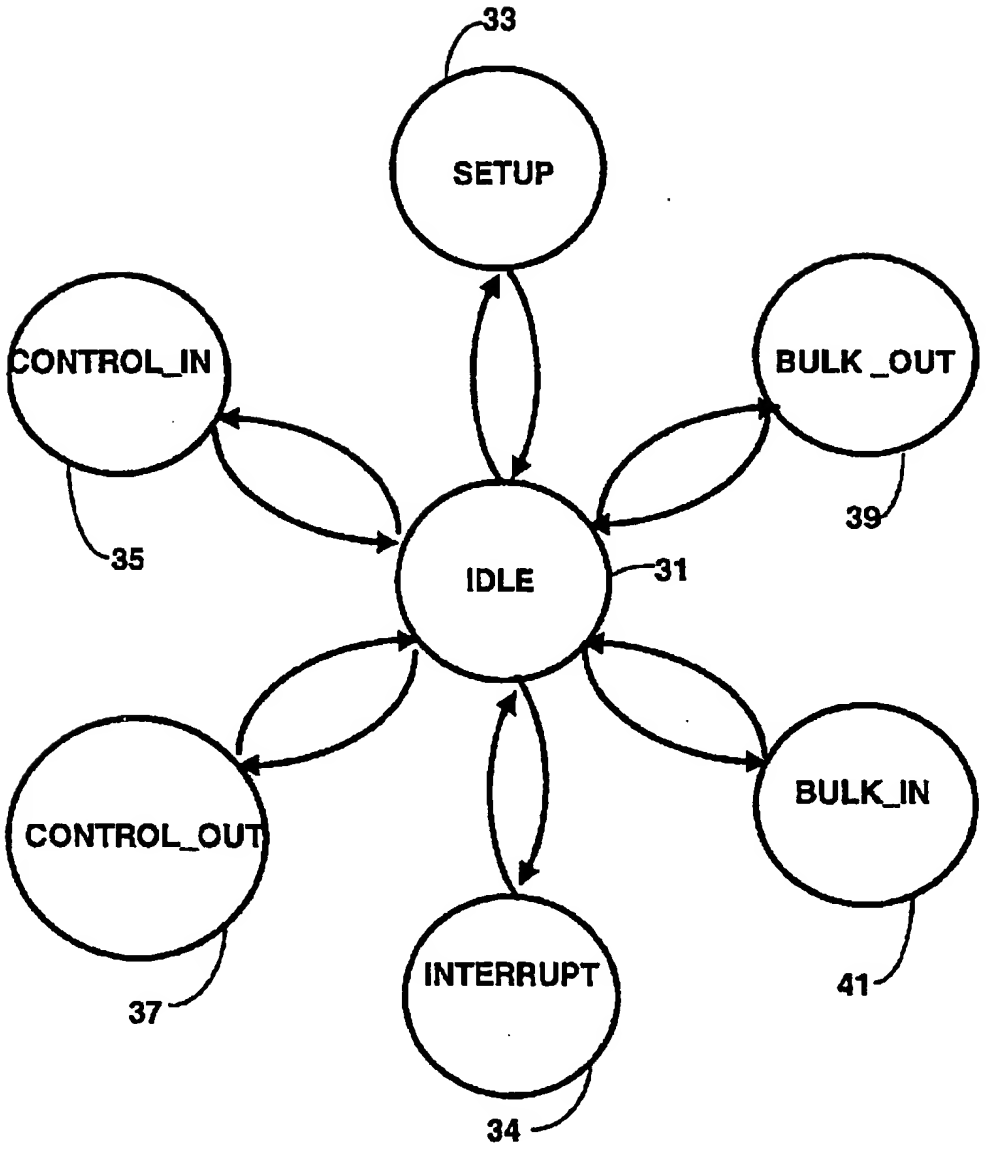

33
SETUP
CONTROL_IN
BULK _OUT
35
39
IDLE
31
CONTROL_OUT
BULK_IN
INTERRUPT
37
41
34

FIG. 3

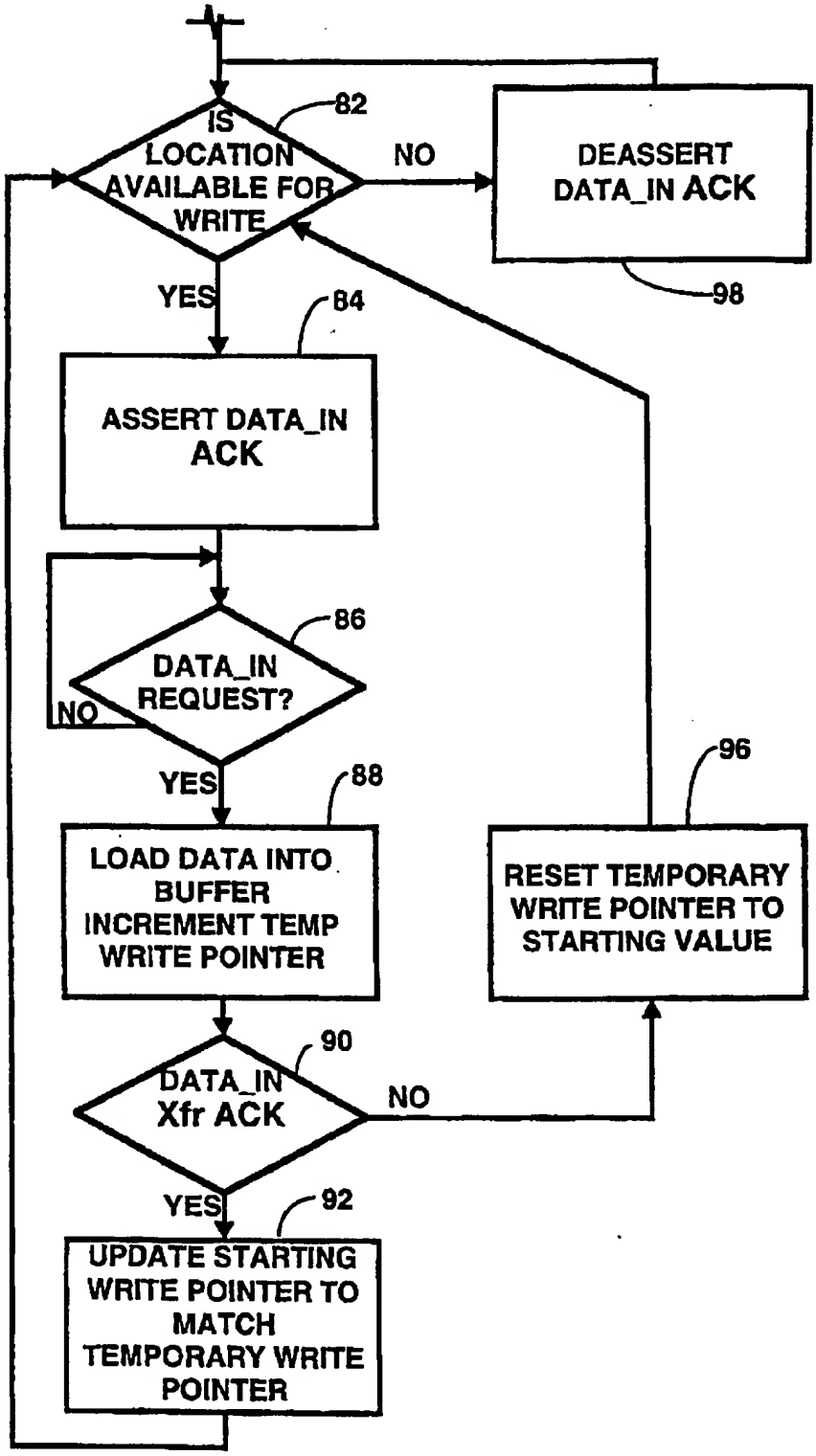

IS LOCATION AVAILABLE FOR WRITE
82
NO
DEASSERT DATA_IN ACK
98
YES
84
ASSERT DATA_IN ACK
86
DATA_IN REQUEST?
NO
YES
88
LOAD DATA INTO BUFFER INCREMENT TEMP WRITE POINTER
96
RESET TEMPORARY WRITE POINTER TO STARTING VALUE
90
DATA_IN Xfr ACK
NO
YES
92
UPDATE STARTING WRITE POINTER TO MATCH TEMPORARY WRITE POINTER

FIG. 4

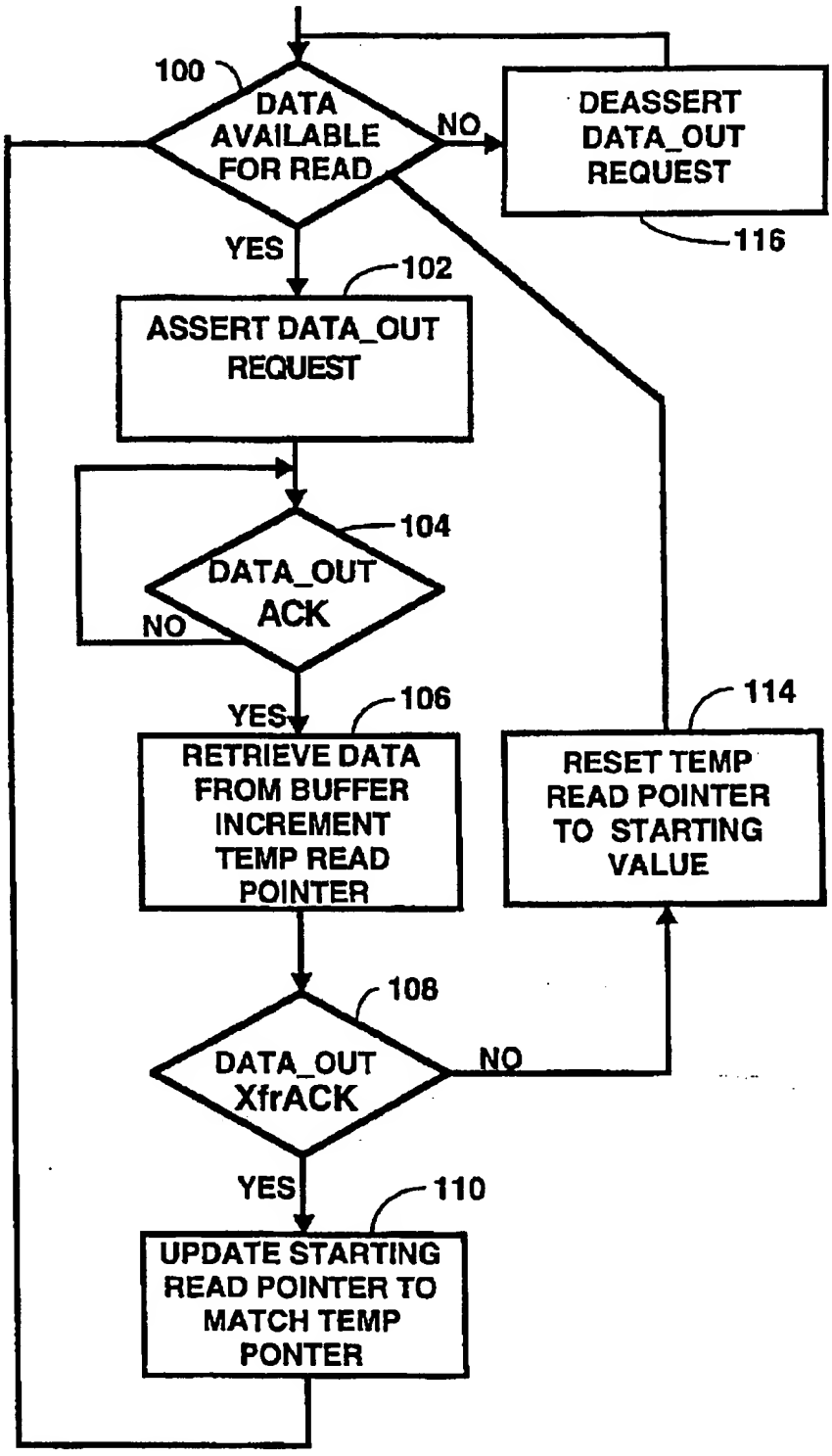

100
DATA AVAILABLE FOR READ
NO
DEASSERT DATA_OUT REQUEST
116
YES
102
ASSERT DATA_OUT REQUEST
104
DATA_OUT ACK
NO
YES
106
RETRIEVE DATA FROM BUFFER INCREMENT TEMP READ POINTER
114
RESET TEMP READ POINTER TO STARTING VALUE
108
DATA_OUT XfrACK
NO
YES
110
UPDATE STARTING READ POINTER TO MATCH TEMP PONTER

FIG. 5

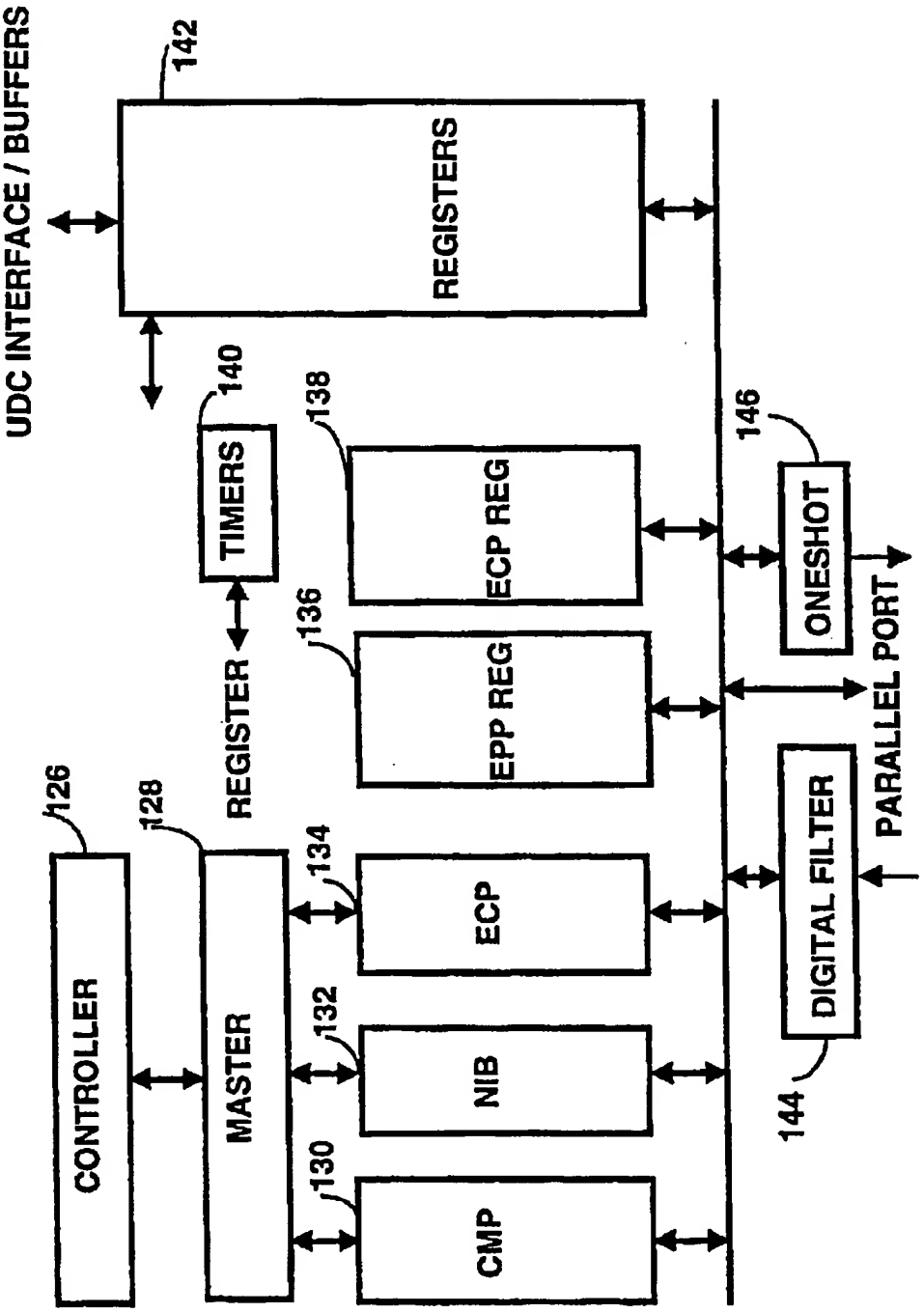

UDC INTERFACE / BUFFERS
142
REGISTERS
140
TIMERS
138
ECP REG
146
ONESHOT
REGISTER
136
EPP REG
PARALLEL PORT
126
128
134
ECP
DIGITAL FILTER
CONTROLLER
MASTER
132
NIB
144
130
CMP

FIG. 6

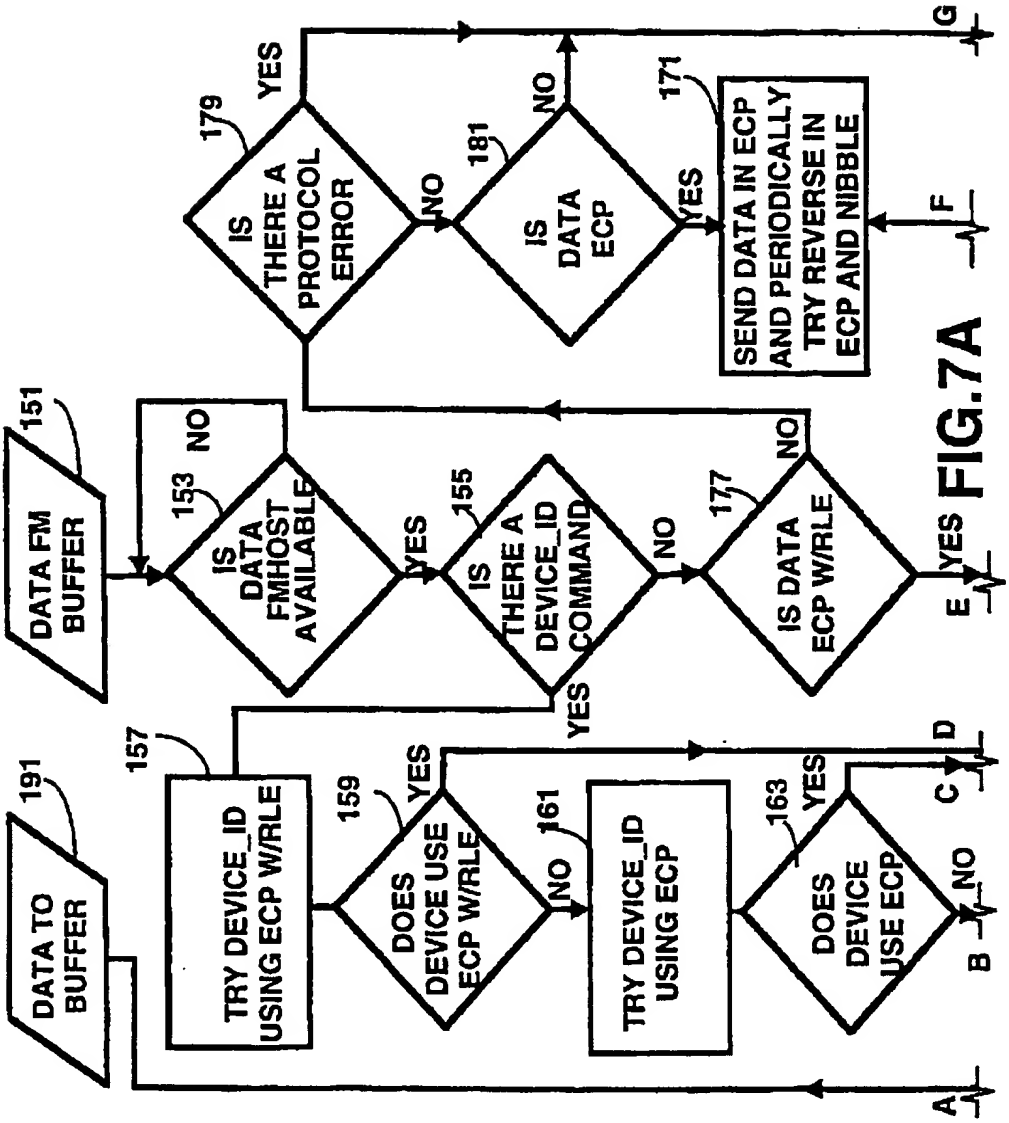

DATA TO BUFFER
191
TRY DEVICE_ID USING ECP W/RLE
157
DOES DEVICE USE ECP W/RLE
159
YES
NO
TRY DEVICE_ID USING ECP
161
DOES DEVICE USE ECP
163
YES
NO
A
B
C
D
DATA FM BUFFER
151
IS DATA FMHOST AVAILABLE
153
NO
YES
IS THERE A DEVICE_ID COMMAND
155
YES
NO
IS DATA ECP W/RLE
177
NO
YES
E
IS THERE A PROTOCOL ERROR
179
YES
NO
IS DATA ECP
181
NO
YES
SEND DATA IN ECP AND PERIODICALLY TRY REVERSE IN ECP AND NIBBLE
171
F
G

FIG.7A

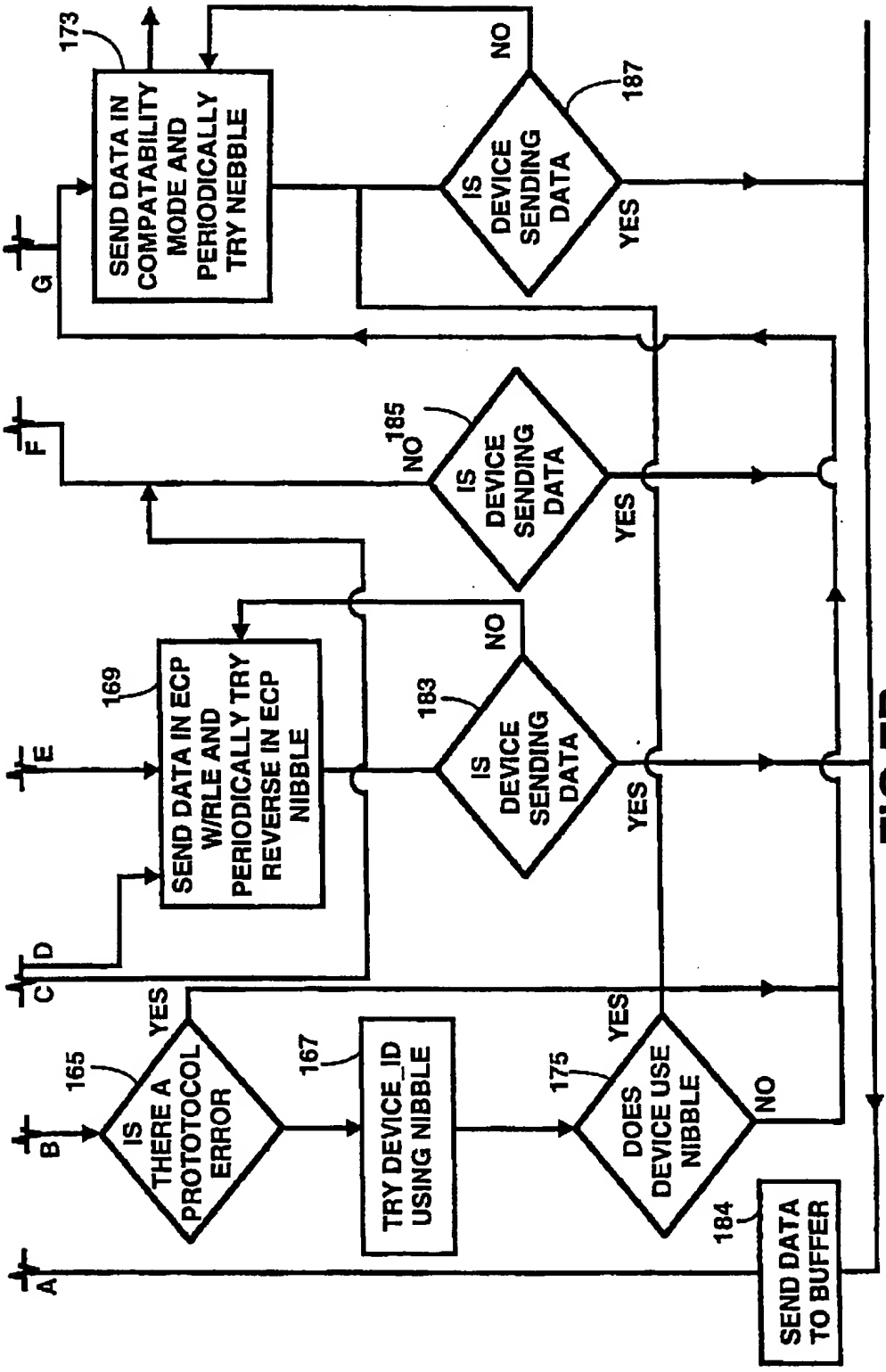

A
B
C
D
E
F
G
165
IS THERE A PROTOTOCOL ERROR
YES
167
TRY DEVICE_ID USING NIBBLE
175
DOES DEVICE USE NIBBLE
YES
NO
184
SEND DATA TO BUFFER
169
SEND DATA IN ECP W/RLE AND PERIODICALLY TRY REVERSE IN ECP NIBBLE
183
IS DEVICE SENDING DATA
NO
YES
185
IS DEVICE SENDING DATA
NO
YES
187
IS DEVICE SENDING DATA
NO
YES
173
SEND DATA IN COMPATABILITY MODE AND PERIODICALLY TRY NEBBLE

FIG.7B

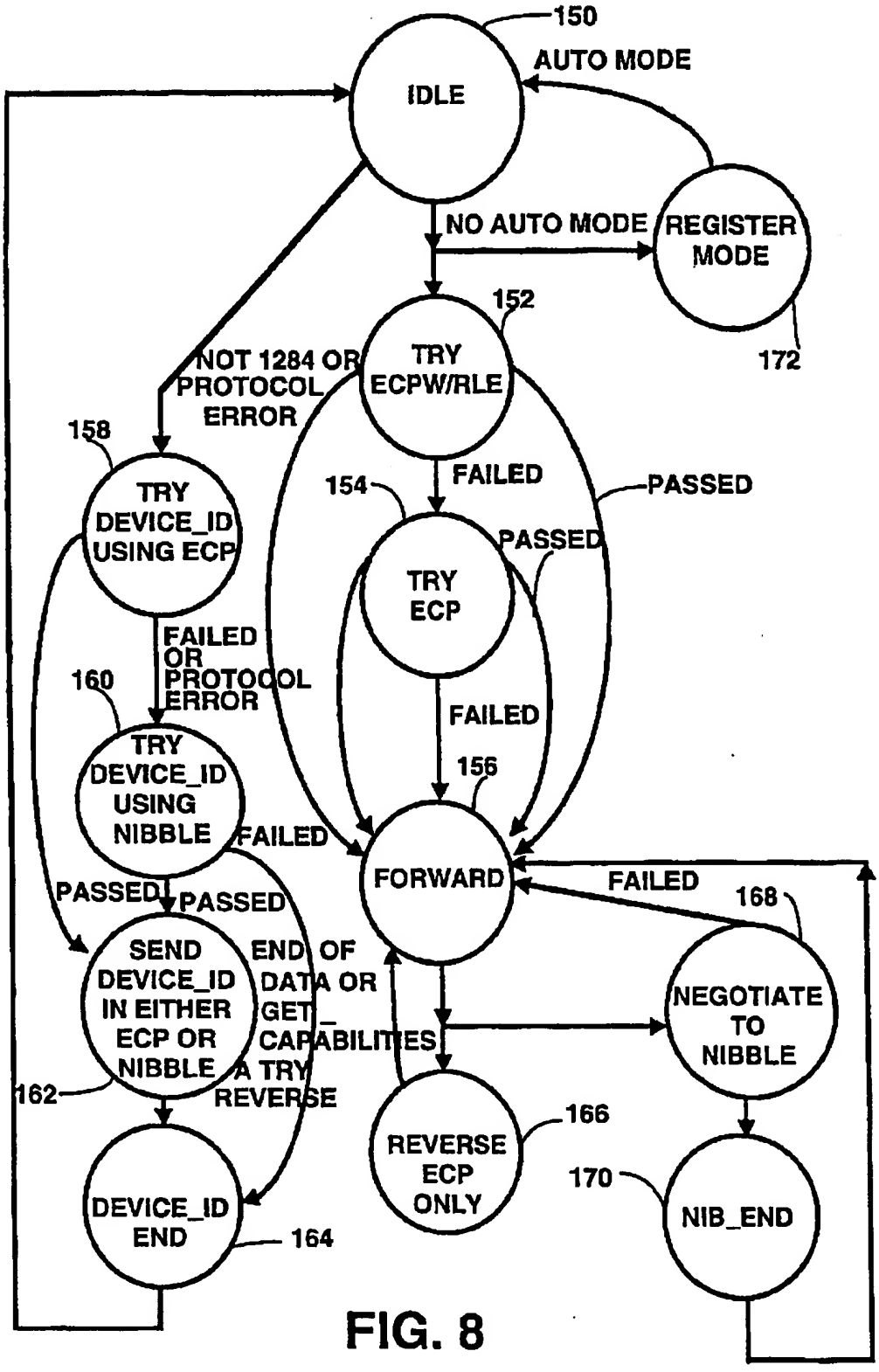

150
IDLE
AUTO MODE
NO AUTO MODE
REGISTER MODE
172
152
TRY ECPW/RLE
NOT 1284 OR PROTOCOL ERROR
158
TRY DEVICE_ID USING ECP
FAILED
154
PASSED
PASSED
TRY ECP
FAILED OR PROTOCOL ERROR
160
TRY DEVICE_ID USING NIBBLE
FAILED
FAILED
156
FORWARD
PASSED
PASSED
FAILED
168
SEND DEVICE_ID IN EITHER ECP OR NIBBLE
END OF DATA OR GET_CAPABILITIES A TRY REVERSE
NEGOTIATE TO NIBBLE
162
REVERSE ECP ONLY
166
170
DEVICE_ID END
164
NIB_END

FIG. 8

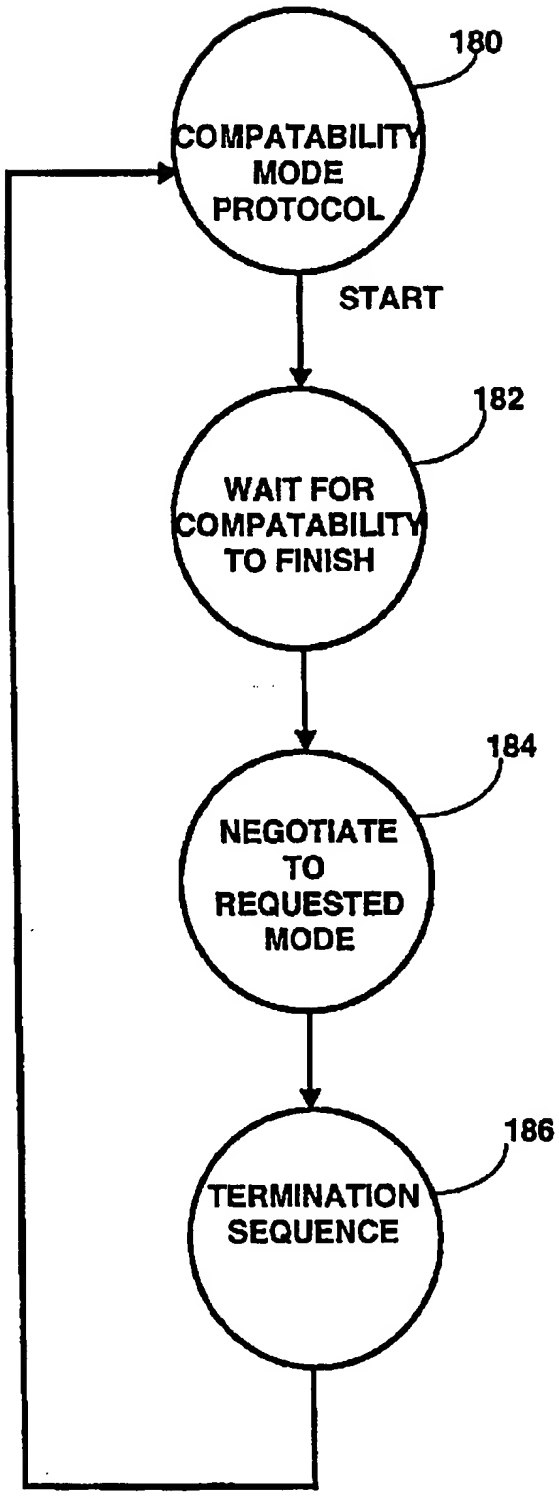

180
COMPATABILITY MODE PROTOCOL
START
182
WAIT FOR COMPATABILITY TO FINISH
184
NEGOTIATE TO REQUESTED MODE
186
TERMINATION SEQUENCE

FIG. 9

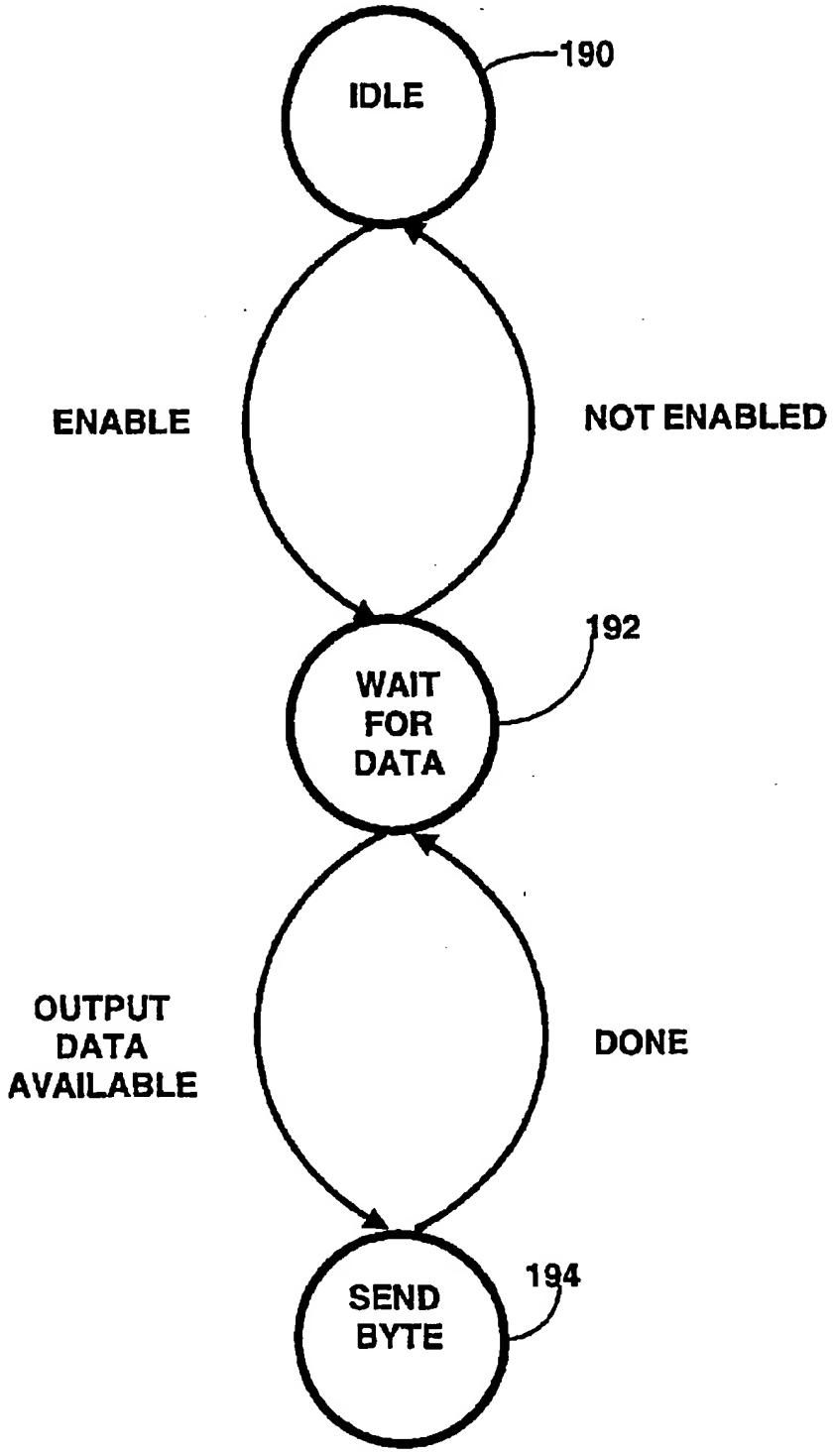

IDLE
190
ENABLE
NOT ENABLED
WAIT FOR DATA
192
OUTPUT DATA AVAILABLE
DONE
SEND BYTE
194

FIG. 10

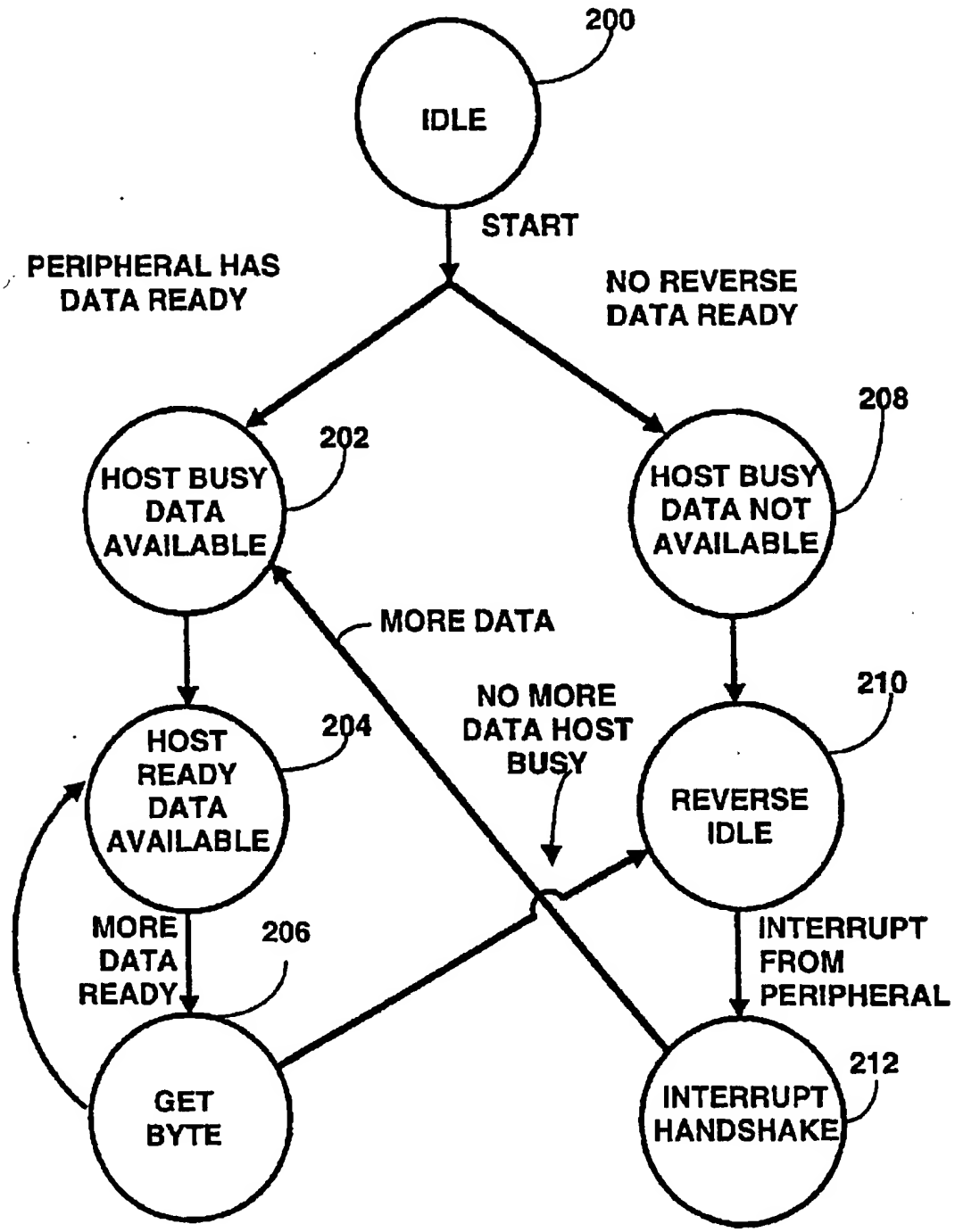

200
IDLE
START
PERIPHERAL HAS DATA READY
NO REVERSE DATA READY
208
202
HOST BUSY DATA AVAILABLE
HOST BUSY DATA NOT AVAILABLE
MORE DATA
210
NO MORE DATA HOST BUSY
204
HOST READY DATA AVAILABLE
REVERSE IDLE
MORE DATA READY
206
INTERRUPT FROM PERIPHERAL
212
GET BYTE
INTERRUPT HANDSHAKE

FIG. 11

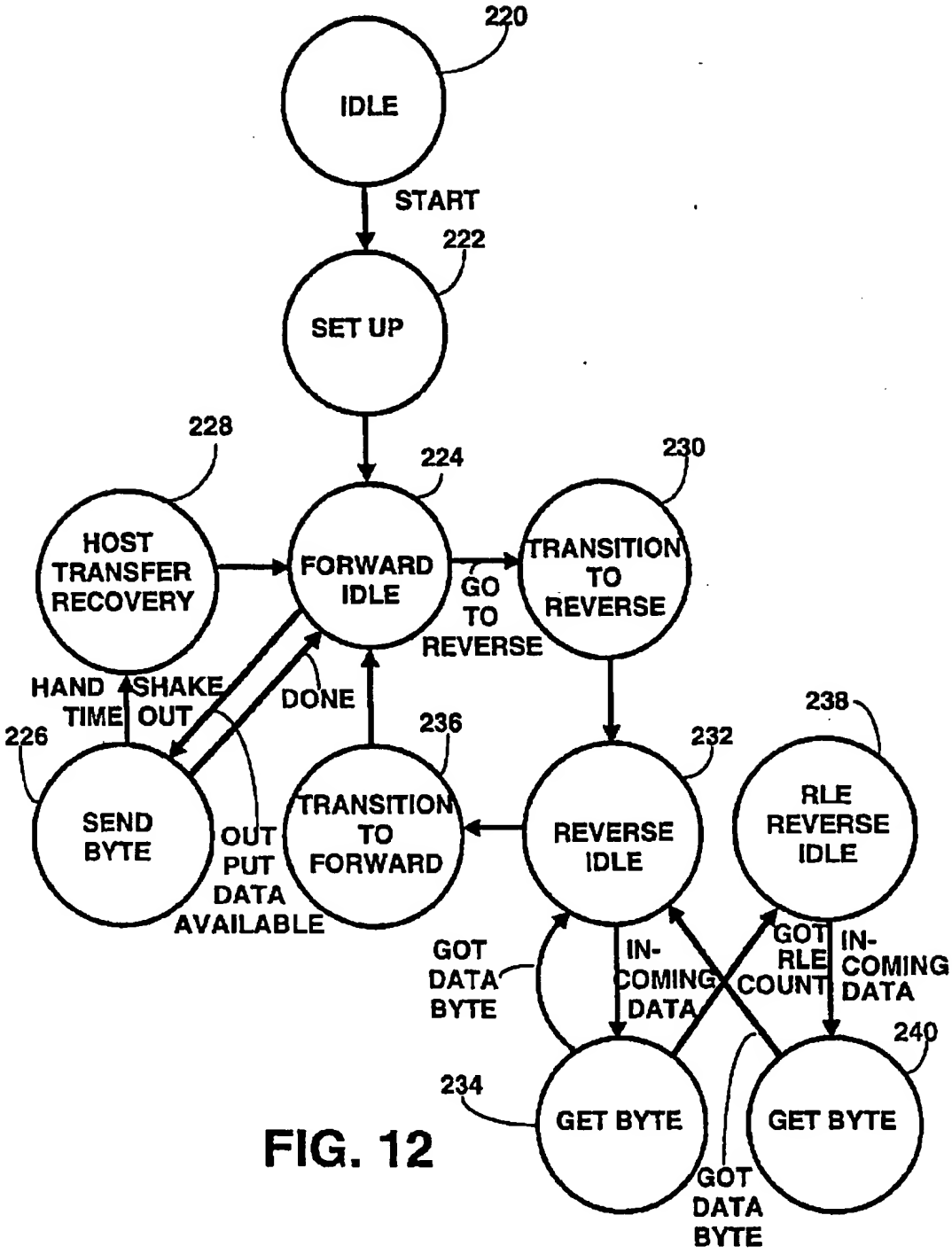

220
IDLE
START
222
SET UP
228
224
230
HOST TRANSFER RECOVERY
FORWARD IDLE
GO TO REVERSE
TRANSITION TO REVERSE
HAND SHAKE TIME OUT
DONE
226
236
238
232
SEND BYTE
OUT PUT DATA AVAILABLE
TRANSITION TO FORWARD
REVERSE IDLE
RLE REVERSE IDLE
GOT DATA BYTE
IN-COMING DATA
GOT RLE COUNT
IN-COMING DATA
240
234
GET BYTE
GET BYTE
GOT DATA BYTE

FIG. 12

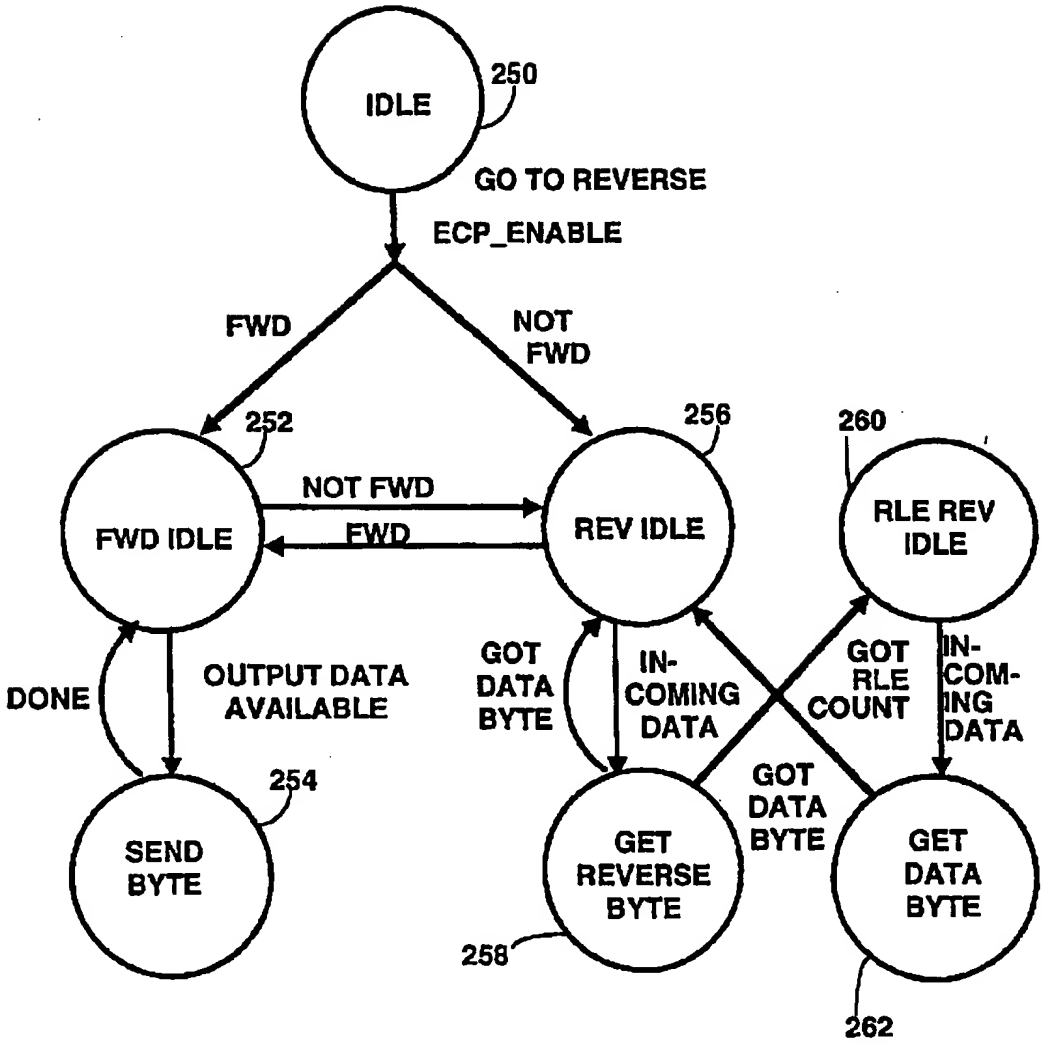

IDLE
250
GO TO REVERSE
ECP_ENABLE
FWD
NOT FWD
252
256
260
FWD IDLE
NOT FWD
FWD
REV IDLE
RLE REV IDLE
DONE
OUTPUT DATA AVAILABLE
GOT DATA BYTE
IN-COMING DATA
GOT RLE COUNT
IN-COM-ING DATA
254
SEND BYTE
GET REVERSE BYTE
GOT DATA BYTE
GET DATA BYTE
258
262

FIG. 13

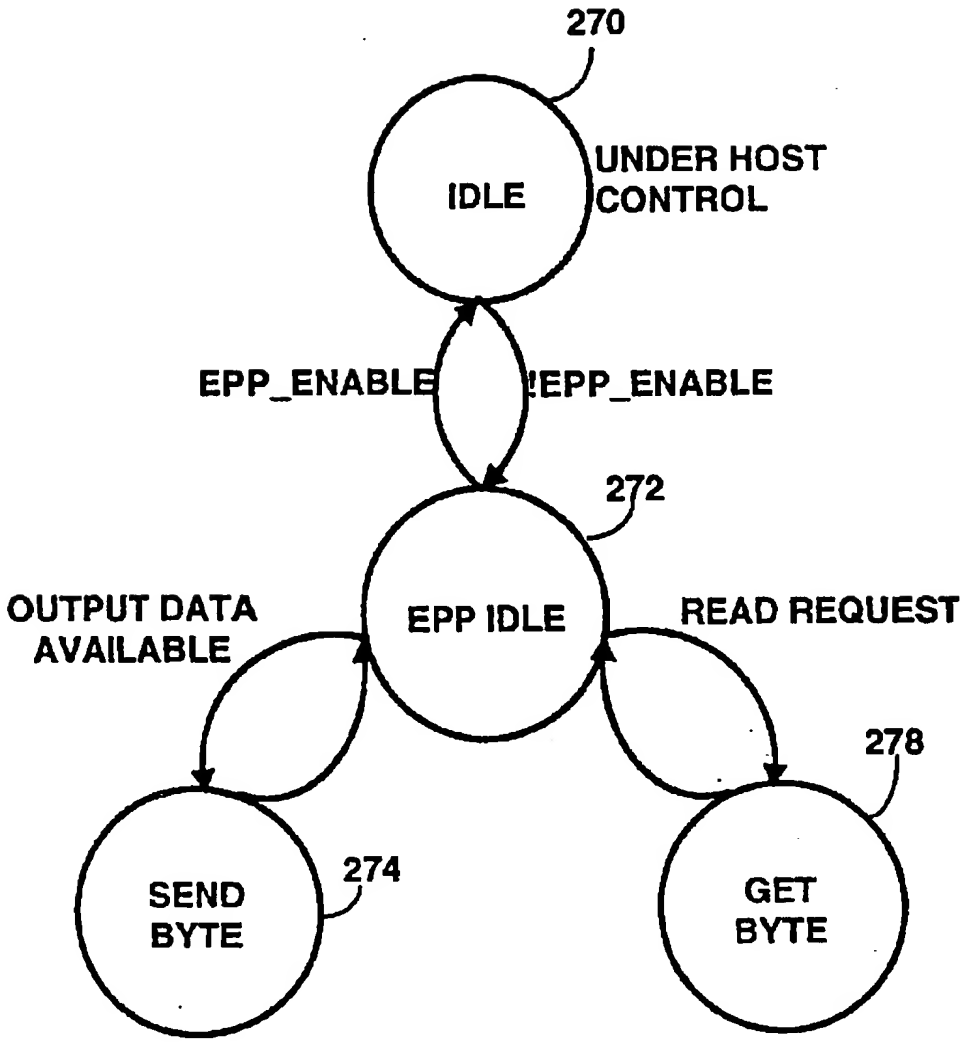

270
IDLE
UNDER HOST CONTROL
EPP_ENABLE
!EPP_ENABLE
272
EPP IDLE
OUTPUT DATA AVAILABLE
READ REQUEST
278
274
SEND BYTE
GET BYTE

FIG. 14

UNIVERSAL SERIAL BUS TO PARALLEL BUS SIGNAL CONVERTER AND METHOD OF CONVERSION

This application is a continuation of Ser. No. 08/974,736 filed Nov. 19, 1997.

BACKGROUND OF THE INVENTION

1. Technical Field

This invention generally relates to a signal converter for converting signals transmitted from a Universal Serial Bus conforming to standards, as implemented by the Universal Serial Bus Implementation Forum, to signals transmitted from a parallel port conforming to IEEE Standard 1284 and the reverse.

2. Background

Throughout its history, the computer and computer peripheral electronics industry has made a continuing effort to standardize input/output ports and signal or communication protocols. This has been, in large part, accomplished by adoption and adherence to industry standards such as those set forth by the Institute of Electrical and Electronic Engineers (IEEE).

A number of computer peripherals, including most printing devices, some paper and photograph scanners, and also some peripheral memory storage devices, are designed for interconnection to a host computer through the standard, and well known parallel port connector which conforms to the IEEE 1284 standard as adopted in the fall of 1994.

With printers, the typical information flow and processing steps can be demonstrated by simple example as follows: The document to be printed is first generated in the host computer using information processing application software, such as a word processing, or a spread sheet, program. The document, in the form of an electronic file of information, is then processed in a second software program, usually called a printer driver, where the information from the original application file is converted into a string of data bits which will ultimately be used by the printer to generate pixels in the complete dithered printed image. The printer driver software will perform such functions as scaling of pixels, addressing, adding color data, and often times even compressing the data in the event of redundant data.

This data stream is then passed through a third, lower level driver software application, usually the operating system software, where it is assembled into bytes suitable for transmission through the host computer's parallel port to the computer peripheral, which in this example is a printer. Over the years, a number of communication protocols have been developed for use in communication between the host computer and the printer peripheral. The earliest and simplest of these protocols is known as the compatibility mode protocol, in which data is sent from the host computer to the printer in one direction only, in eight bit, or one byte, parallel format. A more advanced version of compatibility mode includes what is known as the NIBBLE mode protocol which provides or allows specific information to flow back from the printer to the host computer over dedicated pins of the parallel port, four bits at a time, and enables the printer to report to the host computer status conditions.

Still later, the Extended Capabilities Port (ECP) protocol was developed wherein eight bits, one byte, of information can flow in either direction between the host computer and the peripheral printer. Another protocol, known as the Enhanced Parallel Port (EPP) protocol permits simultaneous transmission of a byte of information in both directions between the host computer and the printer.

The vast majority of printers of whatever type, make and manufacture, that are in use and are currently being manufactured in the United States, are for use with one or more of these communications protocols in conjunction with a parallel port conforming to the IEEE 1284 standard.

Other computer peripherals, including document scanners and peripheral memory storage devices, also utilize the parallel port conforming to the IEEE 1284 standard, and these communications protocols. However, because of the differing requirements, there may be additional protocols built into the peripheral driver resident in the host which are not standard in compatibility mode, enhanced capabilities port mode, or in the extended parallel port mode.

The Universal Serial Bus, USB, communication protocol is different in some fundamental areas, not the least of which is USB is a serial communications protocol, which is designed around shift registers. As a result, it is not possible to connect the input/output, I/O, USB port to a device designed to receive and transmit data through an I/O parallel port conforming to the IEEE 1284 standard. Accordingly, what is needed is a device which can be used to connect a USB ported host to a peripheral as a parallel ported, IEEE 1284 conforming, host. It is one object of this invention to provide a converter which can operate in an automatic mode as a fully compliant USB device receiving and sending data using USB communications protocols, and as a re-transmitting device sending and receiving data to an attached peripheral device as a fully compliant parallel port device, all done in a transparent manner wherein that the host need have no knowledge that the protocol translation is occurring.

It is another object of this invention to provide a signal converter which can operate in a register mode wherein the signal converter contains a set of registers which emulate those found in standard computer parallel port hardware.

SUMMARY OF THE INVENTION

These objects are accomplished in a serial to parallel port signal converter which is preferably manufactured as a one-chip serial to parallel port signal converter which converts a bit stream signal coming from the universal serial bus of a host device to a parallel signal conforming to the Institute of Electronic and Electrical Engineers (IEEE) 1284 signal protocol. It is connected to, and appears to a universal serial bus (USB) of a host, as a standard USB device, and to a peripheral device as an IEEE 1284 host. It operates in two different modes, the first being the automatic mode wherein it acts as a fully compliant bi-directional USB device receiving USB data packets and retransmitting that data to the attached peripheral device transparently as if it were an IEEE 1284 host. In automatic mode, the actual host device has no knowledge that the protocol translation is occurring.

In the second mode, register mode, the signal converter contains a set of registers which emulate those found in standard, IEEE compliant parallel port hardware.

The serial to parallel port signal converter can be understood representationally as a series of modules, the first being the universal serial bus device controller module which is connected to the universal device controller interface, a buffer memory, a read-only memory, and a parallel port interface module. The universal serial bus device controller is an application specific standard product developed by Sand Microelectronics, Inc., and available from Lucent Technologies and is known by the macrosell

name of UDC as published in the Lucent Technologies System ASIC data book dated September, 1996. It functions as a controller for managing signals to and from the universal serial bus of the host, including generation and transmission of start codes, data strobes, and control and data signals. It handles most of the low level USB protocol operations and converts USB bit streams of data to a stream of bytes, plus control and status information. It is used to separate the cyclical redundancy check signal from the data, while it keeps the data in a register and verifies the accuracy of the cyclical redundancy check signal, and is used to send acknowledgment or non-acknowledgment signals to a universal device controller interface. The universal serial bus device controller also performs an additional function in that it stores the information about the end points of data streams and the devices supported by the serial to parallel port signal converter.

A universal device controller interface is provided and is utilized as a control device for the universal serial bus device controller, adding support for the USB protocol features not handled directly by the universal serial bus device controller. Some of these additional features include decoding of descriptor requests, and providing of descriptor data, which is sourced by the read-only memory by way of a buffer memory. It also provides support for all printer class device commands and support which is unique and specific to the present invention and not part of any standard specification. The universal device controller interface has ports for communicating to the buffer memory and also a port for communicating directly with the parallel port interface module.

A buffer memory is provided and is a block which has byte-oriented storage for multiple data packs, which in the preferred embodiment are packets of sixty-four (64) bytes. There are independent input and output channels for two packets of one hundred twenty eight (128) bytes of actual value in each direction.

The parallel port interface module is made of hardware for a fully automatic support of Institute of Electronic and Electrical Engineers (IEEE) 1284 standardized compatibility, NIBBLE, extended capability port and enhanced parallel port with and without run length encoded communications modes. It is intended to emulate normal parallel port hardware such as might be implemented with an industry standard super-I/O chip. It also includes logic to provide control for the automatic and register based parallel port logic, including readable and writeable registers which emulate those found in standard personal computers. It also includes necessary support logic, such as timers, counters, digital filters and post stretchers.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a high level block diagram of the serial bus to parallel bus signal converter;

FIGS. 2A through 2D is a flow chart block diagram of the universal device controller interface;

FIG. 3 is a state transition diagram for the universal device

FIG. 4 is a block diagram flow chart of the data write operations of the buffer;

FIG. 5 is a block diagram flow chart of the data read operations of the buffer;

FIG. 6 is a high level block diagram of the parallel port interface module.

FIG. 7 is a flow chart block diagram of the parallel port interface module;

FIG. 8 is a state transition diagram for the controller operations of the parallel port interface module;

FIG. 9 is a state transition diagram for the master state machine operations of the parallel port interface module;

FIG. 10 is a state transition diagram of the operations f the compatibility mode host for the parallel port interface module;

FIG. 11 is a state transition diagram of the NIBBLE mode host of the parallel port interface module;

FIG. 12 is a master state transition diagram of the ECP host operations of the parallel port interface module;

FIG. 13 is a diagram of the ECP host in register mode operations of the parallel port interface module;

FIG. 14 is a state transition diagram of the EPP host operations in register mode operations for the parallel port interface module.

DESCRIPTION OF THE PREFERRED EMBODIMENT

The serial to parallel port signal converter 10 of the present invention is shown in a high-level block in FIG. 1. It is preferably manufactured as a one-chip serial to parallel port signal converter which converts a bit stream signal coming from the universal serial bus of a host device to a parallel signal conforming to Institute of Electronic and Electrical Engineers (IEEE 1284) signal protocol. It is connected to, and appears, to a universal serial bus of a host, hereinafter USB, as a standard USB device, and to a peripheral device as an IEEE 1284 host. It operates in two different modes, the first being the automatic mode, wherein it acts as a fully compliant bi-directional USB device receiving USB data packets and retransmitting that data to the attached peripheral device transparently as if it were a IEEE 1284 host. In automatic mode, the actual host device has no knowledge that the protocol translation is occurring.

In the second mode, register mode, the signal converter contains a set of registers which emulate those found in standard, IEEE 1284 compliant parallel port hardware.

As shown in FIG. 1, the serial to parallel port signal converter 10 can be shown representationally as a series of modules, the first being the universal serial bus device controller module 12 to which is connected the universal device controller interface 14, buffer memory 16, read only memory 18, and parallel port interface module 20. The universal serial bus device controller 12 is, in the preferred embodiment, an application specific standard product developed by Sand Microelectronics, Inc., and available from Lucent Technologies®. and is known by the MACROCELL name of UDC as published in the_Lucent Technologies System ASIC Data Book dated September, 1996.

Universal serial bus device controller 12 is known in the prior art and its functions play no part of the present invention, except as a controller for a managing signals to and from the universal serial bus of the host in the manner which is hereinafter generally described.

Universal serial bus device controller 12 is used in the present invention to generally manage the USB data transmission, either to or from the host, including generation and transmission of start codes, data strobes and control and data signals. It handles most of the low level USB protocol operations and converts USB bit streams of data to a stream of bytes, plus control and status information. It is used to separate the cyclical redundancy check signal from the data, while it keeps the data in a register and verifies the accuracy of the cyclical redundancy check signal, and is used to send

acknowledgment or non-acknowledgment signals to the universal device controller interface 14.

This is accomplished in the universal serial bus device controller 12 by a series of blocks, not shown in the diagrams, which include a phase lock loop for synchronizing the clock signals of universal serial bus device controller 12 and the entire serial to parallel port signal converter 10 to the clock signals of the host, and a serial interface engine which does the initial functions of the USB protocol: syncField identification, NRZI-NRZ conversion, token packet decoding, bit stripping, bit stuffing, NRZ-NRZI conversion, CRC5 checking and CRC 16 generation and checking. It also converts the serial packets from the USB signal to 8 bit parallel data. The universal serial bus device controller 12 also performs an additional function in the preferred embodiment, in that it stores the information about the end points and the devices supported by the serial to parallel port signal converter 10.

The universal device controller interface is utilized as a control device for the universal serial bus device controller, adding support for the USB protocol features not handled directly by the universal serial bus device controller 10. Some of these additional features include decoding of descriptor requests and providing of descriptor data, which is sourced by the ROM 18 by way of buffer memory 16, and also provides support for all printer class device commands and support which is unique and specific to the present invention and not part of any standard specification. Universal device controller interface 14 has ports for communicating to the buffer memory and also a port for communicating directly with the parallel port interface module 20.

Buffer memory 16 is simply that, in that it is a block which has byte-oriented storage for multiple data packets, which in the preferred embodiment are packets of sixty four bytes. There are independent input and output channels, with buffering for two packets of 128 bytes of actual value in each direction.

Read only memory, ROM, 18 stores the specifications and information necessary to convert descriptor data from one communications protocol to another, as is hereinafter described.

The parallel port interface module 20 is made up of hardware for fully automatic support of Institute of Electrical and Electronic Engineers (IEEE) 1284 standardized compatibility, NIBBLE, extended capability port (ECP) and enhanced parallel port (EPP) with and without run length encoded (RLE) communications modes, and is intended to emulate normal parallel port hardware, such as might be implemented with an industry-standard super-I/O chip. It also includes logic to provide control for the automatic and register based parallel port logic, including readable and writeable registers which emulate those found in standard personal computers. It also includes necessary support logic such as timers, counters, digital filters and pulse stretchers, as is hereinafter described.

FIGS. 2A through 2D, a flow diagram, illustrate the actions taken in the universal device controller interface 14. In the first step, a data input is initiated in block 30 from the universal serial bus device controller 12, after which in decision block 32, a decision is made as to whether a data transfer has been initiated. If no data transfer is initiated, the universal device controller interface continues to wait for the initiation of a transfer, as is also shown in circle 31 of state transition diagram, FIG. 3.

Once a transfer is initiated, the transfer is decoded in Block 34 to identify the type of data being transmitted, as either being a command packet, in which case it is transferred to block 36, and interrupt request packet which is ported to block 38, or a data packet which is sent to block 40.

As shown in FIG. 3, once a transfer is initiated, universal device controller 14 transitions to a set up state as shown in circle 31 wherein the data from the USB host coming from_universal serial bus controller is loaded into an application bus. The completion of the receipt of data acknowledgment signal (ACK) or a not-acknowledged signal (NACK) transitions universal device controller interface 14 back to the idle state as shown in circle 31. If the data loaded during the set up, shown in circle 33, is a command packet, the universal device controller interface 14 transitions to the CONTROL_IN state as shown in circle 35 as the command is decoded. And its transfer is either acknowledged or not acknowledged. If it is not acknowledged, the universal device controller interface returns to idle state 31.

If the data is a command packet, as shown in FIG. 2B, the command packet is then decoded in block 42 and, in decision block 44, a decision is made as to whether it is a valid transfer of command packet data. In this decision block, an acknowledgment signal (ACK) or a not acknowledged signal (NACK) is received from the universal serial bus device controller, either acknowledging or not acknowledging the receipt of the valid cyclical redundancy check, (CRC) signal. If the decision is no, the proper acknowledgment has not been received, then the universal device controller interface resets to decision block 32 to await a valid transfer initiation.

If a valid acknowledgment is received, as determined by decision block 44, then a decision is made as to whether or not the command signal indicates that the host device requires that data be sent back to the host device regarding status information of the peripheral to which the parallel port interface module 20 is connected. This is also shown in the state transition diagram of FIG. 3 as state 37. If the answer is no, no status data is required, or no CONTROL_OUT transfer is received a decision is made in decision block 48, to send a stall handshake. If the decision is yes, and the signal has been received, then in block 62 a signal of a valid transfer is initiated back to the universal serial bus device controller 12. If the answer in decision block 48 is no, that is to say that a CONTROL_IN was not expected, then in box 50, there is sent a stall handshake back to the universal serial bus device controller 12 and the universal device controller transitions back to idle state shown as Circle 31 in FIG. 3.

If, as determined in decision box 46, data is required by the host, then, in decision box 52, the universal device controller interface 14 awaits for the receipt of a CONTROL_IN transfer indicating that the host is ready to receive the requested status data from the peripheral device as it transitions to the CONTROL_IN_state of circle 35 of FIG. 3. As shown in block 54, the data is sent to the host and the universal device controller interface 14 awaits, in decision block 56, an acknowledgment that the transfer was received by the host. If, in decision block 52, it is determined that no_CONTROL_IN transfer is received, or a CONTROL_OUT transfer is received, then as shown in box 50, a stall handshake is sent in the same manner as if a CONTROL_IN transfer was not included in the status for a command not requiring return data or the status transfer in the command was a CONTROL_OUT.

Once the data has been sent to the host in state 39 of FIG. 3_and in block 54 of FIG. 2, universal device controller

interface 14 awaits, in decision block 56, receipt of an acknowledgment of the transfer. If acknowledgment is received, in block 62, the universal device controller interface signals a valid transfer. If the transfer acknowledgment is not received, then as shown in block 60, the device signals an invalid transfer.

As shown in state circle 43 of FIG. 3, and in FIG. 2C, if the data identified in block 34 is an interrupt request packet in block 38 of FIG. 2A, a decision is made in decision block 50 whether the requested data is available. If the answer is yes, as shown in circle 41 of FIG. 3 and then in block 52 of FIG. 2 the data is sent through the universal serial bus device controller 12 to the host and then awaits, in decision block 54, for an acknowledgment of receipt of the transferred data.

If the data transfer is acknowledged, the universal device controller interface then signals that a valid transfer has occurred and resets in decision block 60. If in decision block 50 it is determined that the requested data is not available, then a not acknowledged handshake packet signal is sent to universal serial bus device controller 12 and ultimately to the host. If the requested data was available and was sent in box 52, and no acknowledgment is received in decision box 54, then as is shown in box 58, the universal device controller interface 14 signals an invalid transfer and the device resets.

If the data transfer initiation identified in box 34 indicates that it is a data packet, then in decision box 62, as shown in FIG. 2D, a decision is made as to whether it is a BULK_OUT packet of data from the host or a BULK_IN data packet to be sent from the universal device to the host in which case universal device controller interface transitions to the BULK_OUT_or the BULK_IN state shown in circles 39 and 41 of FIG. 3, as the case may be.

If it is a BULK_OUT packet, and the decision in decision box 62 is no, then in decision box 64 the decision is made as to whether there is sufficient storage space in the buffer to receive the data. Data packets are transferred through the universal serial bus device controller 12 in eight bit bytes, one at a time. As each byte is received, the decision is made concerning space availability in decision box 64. If the decision is yes, then the data is sent to the buffer for storage, as shown in box 66. If the decision is no, a not acknowledged handshake packet is sent as shown in box 68. Once the data is received and sent to the buffer in box 66, a decision is made in decision box 70 as to whether an acknowledgment is received for the data. If the answer is yes, then in box 72 the universal device controller interface 14 signals a valid transfer.

If the decision in decision box 62 is that the data is a BULK_IN packet which is data received from the peripheral for transfer to the host in response to an interrupt request, then in decision box 74, a decision is made as to whether the data is available in the buffer. If it is not, then a not acknowledged (nACK) handshake packet is sent to the universal serial bus device 12. If the answer is yes, then in box 76 the data is retrieved from the buffer, and in box 78 is sent to the host, and in decision box 70 a decision is made as to whether or not an acknowledgment is received for the data sent. If the answer is yes, then in box 72 the interface 14 signals a valid transfer and if the answer is no, then as shown in box 80, the signal is for an invalid transfer.

FIG. 4 is a flow chart showing how data is written into buffer memory 16 in response to the operation shown in FIG. 66 of FIG. 2D when data is sent to the buffer.

As data is sent to buffer memory 16 as shown in block 66 of FIG. 2D, the first decision made in buffer memory 16 is as shown in decision box 82 of FIG. 4. The decision made is whether there is a location available for writing the next byte of information. In the buffer, in the preferred embodiment, each byte of information is treated and processed separately. If the answer in decision box 82 is yes, there is a location available to write a byte of information, then in box 84 an ASSERT DATA_IN signal is sent to universal device controller interface 14 acknowledging that there is a space available. Once the signal is sent, then buffer memory 16 waits, at decision box 86, to receive the DATA_IN request. Once the DATA_IN request is received as shown in decision box 86, the operations described in box 88 occur, namely the data is loaded into buffer memory 16, and the temporary write pointer is incremented to the next byte in the buffer memory and buffer memory 16 then waits for an acknowledgment that the transfer is completed in decision box 90, and if it is received, then as shown in box 92, the starting write pointer is updated to match the temporary write pointer.

In the event no transfer acknowledgment is received regarding the DATA_IN signal, then in decision box 94 a not acknowledged (nACK) signal regarding receipt of the DATA_IN signal. After which, in box 96 the temporary write pointer is reset to the original starting value.

In the event no location is available for writing the byte of data, then as shown in box 98, a DE_ASSERT data input acknowledgment signal is sent.

How data is read from buffer memory 16 is shown in the flow chart of FIG. 5. First, in decision box 100, a decision is made as to whether the data byte is available for read. If the answer is yes, then in box 102 an ASSERT DATA_OUT request is made, and in decision box 104 a decision is made as to whether or not the ASSERTED DATA_OUT request is acknowledged. If not, buffer memory 16 waits for its receipt. Once it is received, then in box 106 the data is retrieved from the buffer and the temporary read pointer is incremented to the next byte. Afterwards, in decision box 108, the data out transfer acknowledgment is awaited, and upon receipt, in box 110, the starting read pointer is updated to match the temporary read pointer.

If the data out transfer acknowledgment is not received, then in decision box 112 a data out transfer is not acknowledged, nACK, is sent and in box 114 the temporary read pointer is reset to its starting value.

In the event that data is not available for read as determined in decision box 100, then a DE_ASSERT DATA_OUT request is made in box 116.

In FIG. 6 there is shown and described a high level block diagram for the parallel port interface module 20. At the heart of parallel port interface module 20 is controller 126, which is used to control, through master state machine 128, compatibility mode protocol host 130, NIBBLE protocol host 132 and extended capabilities port host 134. A digital filter 144 5 is provided for incoming signals. It is optionally used to filter spurious or false signals by holding and not passing on each signal until a predetermined number of clock ticks occur, thus assuring that all signals inputted to parallel port interface module 20 are true signals. Also provided, as shown in the block diagram of FIG. 6 there is an extended parallel port register host 136 and extended capabilities port register host 138. Control registers 142 are provided. One shot 146 generates a fixed pulse signal to transition the parallel port interface module 20 output to high drive by generation of a fixed pulse signal.

FIGS. 7A and 7B represent a flow chart for the operations of_parallel port interface module 20, as shown in FIG. 6. The parallel port interface module 20 attempts to communicate with the peripheral device in the highest, or most advanced, communication protocol that the peripheral device is capable of using. It uses a hierarchle order of the IEEE 1284 communications protocol, starting with the most advanced communications protocol, and stepping down

through hierarchle until it establishes communications with the peripheral device.

As seen in FIGS. 7A and 7B, data from the buffer 16 is received through the parallel port interface module 20 through data transfer box 151. As it is received, the first decision is made in decision box 153 is whether data from the host is available. If the answer is no, then parallel port interface module 20 remains in an idle state. If the answer is yes, then a decision is made in decision box 155, is there a DEVICE_ID command in the data that is available. The DEVICE_ID command is the result of the translation of a USB command protocol entitled GET DEVICE_ID which is the USB command protocol command which requests data from the peripheral regarding the identification of the manufacturer of the peripheral device, the type of device it is, and what communications protocols the peripheral supports. The GET DEVICE_ID command is very similar to the DEVICE_IN command of the extended communication port communications protocol utilized by devices which are compliant with IEEE 1284 specifications. As a result, as shown in FIGS. 2A through 2D, the USB GET DEVICE_ID command of the USB protocol is translated into a DEVICE_ID command. If the answer in decision box 155 is yes, then in box 157 an attempt is made to send the DEVICE_ID command to the peripheral using extended capabilities port protocols with run length encoding. A decision has been made in decision box 159 as to whether the peripheral device communicates in extended capabilities port protocol with run length encoding._If the answer is yes, then in box 169 the command, and later data, is sent in extended capabilities port protocol with run length encoding, and periodically a TRY_REVERSE command is sent in extended capabilities port protocol and NIBBLE. If the answer is no, then in box 161 the parallel port interface module 20 will try the DEVICE_ID command using extended capabilities port protocol without run length encoding. If communication is established in extended capabilities protocol as shown in decision box 163, then in box 171 data is then sent to the device in extended capabilities protocol and the parallel port interface module 20 periodically tries reverse communication in extended_capabilities protocol and_NIBBLE.

If the answer in decision box 163 is no in that the device does not use the extended capabilities communications protocol, _then in decision box 165 a decision is made as to whether there exists a protocol error. If the answer is yes, then data is sent as shown in box 173 in the compatibility mode and the device periodically tries NIBBLE.

If the answer in decision box 165 was no, there was no protocol error, then in box 167 parallel port interface module tries the DEVICE_ID command using NIBBLE. Irrespective of whether the device does or does not use NIBBLE, parallel port interface module 20 will begin to send data in the compatibility mode as shown in box 173 and will periodically try NIBBLE irrespective of whether or not the DEVICE_ID command indicated that the device used it.

If it is determined in decision box 155 that the incoming data does not contain a DEVICE_ID command, then in box 177 a decision is made as to whether the incoming data is enhanced capabilities protocol encoded with run length encoding. If the answer is yes, then the data will be sent as shown in box 169 in extended capabilities protocol with run length encoding and parallel port interface module 20 will periodically try reverse in capabilities protocol and in NIBBLE. If it is determined in decision box 177 that there is no run length encoding, then in decision box 179 the decision is made as to whether or not a protocol error has occurred. If the answer is yes, then the data is sent as shown in box 173 in the compatibility mode. If the decision in box 179 is no, then a determination is made in decision box 181 as to whether the data is in extended capabilities protocol encoded without run length encoding. If the answer is yes, then data is sent in capabilities protocol and the parallel port interface module 20 will periodically try reverse in extended capabilities protocol and NIBBLE. If the decision in decision box 181 is no, then the data is sent in the compatibility mode and the parallel port interface module will periodically try to communicate with the device in NIBBLE.

In each case, whether the data is being sent in extended capabilities protocol with run length encoding, or without it, or if it is being sent in compatibility mode, the parallel port interface module 20 will periodically attempt to run reversed so as to receive data from the device. This is shown in decision boxes 183, 185 and 187. If the decision is yes, then in box 189 the data is sent to the buffer through transfer box 191.

Next, there is shown in FIG. 8 a state transition diagram of the operations of controller 126 of FIG. 6 for the parallel port interface module 20. As previously stated, signal converter 10 can support three of the basic communications protocols used with a parallel port I/O device conforming to the IEEE 1284 standard. These are: the compatibility mode protocol; the extended capabilities port (ECP) with or without run length encoded (RLE) compression protocol; and the enhanced parallel port protocol (EPP). As shown in circle 150 of FIG. 8, the power up state is idle and the parallel port interface module 20 is in compatibility mode. This is the default in which the parallel port interface module 20 will initially start.

After starting in compatibility mode protocol, controller 126 will attempt to change states to the highest level protocol available, which, in the preferred embodiment, in automatic operation is extended capabilities port (ECP) protocol with run length encoding. If the peripheral device will communicate using this protocol, controller 126 passes to the forward state shown in circle 156 wherein the parallel port interface module 20 will communicate and pass data to the peripheral device using extended capabilities port protocol with run length encoding. In the event that the peripheral device cannot communicate in this protocol, then the parallel port interface module 20, controller 126 will attempt to communicate with the peripheral device utilizing the next highest protocol available, which is extended capabilities port protocol as shown in state 154. If the peripheral device can communicate in this state, the machine passes into the forward state shown in circle 156 wherein it again begins sending data to the peripheral device using extended communications port protocol without run length encoding. If the attempt to communicate using extended communications port protocol fails because the peripheral device is not configured to communicate in that protocol, controller 126 shifts into the forward state of 156 utilizing the default protocol, namely the compatibility mode protocol.

If in the attempts to use the higher level protocol languages, controller 126 for the parallel port interface module detects either that the peripheral device is not IEEE 1284 compliant, or that there is a protocol error, it will automatically shift into the forward transmission state shown in circle 156 utilizing the lowest level protocol language, namely the compatibility mode.

One of the initial command packets that will be sent by the host device utilizing USB command protocols is the command GET_DEVICE_ID which is a command which requests data from the peripheral regarding the name of the manufacturer of the peripheral device, the type of device it is, and what communication protocols the peripheral supports. This command is very similar to the DEVICE_ID command utilized by devices which are compliant with IEEE 1284 specifications. As a result, the GET_DEVICE_ ID command f the USB protocol is translated into a

DEVICE_ID command. If the GET_DEVICE_ID command is received, then in parallel port interface module 20, controller 126 will attempt to try the DEVICE_ID and command using extended capability port communications protocol as shown in state circle 158. If it fails in state 158, or if there is a protocol error, it will attempt again to try the DEVICE_ID command in the NIBBLE communication protocol as shown in state 160. If this fails, the controller shifts into forward state 156.

If either the first attempt to communicate the DEVICE_ID command using ECP, (extended capabilities port) or the attempt to communicate it using the NIBBLE protocol succeeds, then controller 126 passes into the state shown in circle 162, namely, the receipt of the DEVICE_ID string from the peripheral device in either Extended Communication Port (ECP) or NIBBLE protocols, whichever first worked. And finally, after receiving the strings, controller 126 enters a state shown in circle 164, namely the signaling of the end of the DEVICE_ID transfer, and a return to the default state of 150, namely the idle state.

During forward state operations as shown in circle 156, when operating in ECP, the controller will periodically issue a TRY_REVERSE command to shift operations into reverse extended capabilities communications protocol, as shown in state 156. This attempt to try reverse operations occurs at a possible end of the data, or after a certain period of time of being in the forward state, in which case the controller operates the parallel port interface module 20 in reverse to pass data from the peripheral back to the host. If forward state operations 156 are being conducted in compatibility mode protocol, any attempt to communicate with the peripheral will be done through negotiation to NIBBLE, as shown in circle 168. If the attempt to negotiate to NIBBLE as shown in 168 fails, then controller 126 reverts to the forward operation of state 156. In the event that the negotiation to NIBBLE of state 168 is successful, there will be NIBBLE communication from the peripheral to the host until the end of the data at which time the NIB_END state shown in circle 170 is extended, and operations of the controller revert to the forward state 156.

The master state machine 128 shown in FIG. 6, is also shown in FIG. 9. It is the state machine that accomplishes the attempt to shift modes first into extended capabilities port protocol with run length encoded data, as shown in state 152 of FIG. 8, and if that fails, then tries to shift into enhanced capability port protocol without run length encoded data, as shown in state 154 of FIG. 8, and finally into compatibility mode protocol for forward operations as shown in state 156 of FIG. 8. It also accomplishes the transitions to negotiations to the NIBBLE protocol and the DEVICE_ID protocol, also as shown in FIG. 8.

As shown in FIG. 9, the starting, or default state 180 is for operation in the compatibility mode protocol. Upon receipt of a start signal, the master state machine waits for if compatibility mode protocol operations to finish in state 182, and in state 184 will negotiate to the requested mode and upon completion of that mode shifts to termination sequence shown in state 186.

FIG. 10 is the state transition diagram for compatibility mode protocol controller 130. Its default state 190 is idled. When enabled, it passes into a wait-for-data state 192, and when output data is available, it passes into send byte state 194.

FIG. 11 is a state transition diagram for NIBBLE controller 132. As shown in FIG. 9, the default state 200 is idle. When controller 126 initiates NIBBLE controller 132, and the peripheral has data ready, NIBBLE controller 132 transitions to state 202, which is defined under IEEE 1284 standards as host busy data available. When the host is ready, the NIBBLE controller transitions to state 204, host ready data available, and then state 206, wherein a byte of data is retrieved. If there is more data to be sent from the peripheral, then the NIBBLE controller transitions back to state 204.

State 208 exists when the host is busy and data is not available, in which case the NIBBLE controller transitions to reverse idle state 210 until it receives an interrupt from the peripheral indicating data is available, as shown in state 212, in which case the NIBBLE controller 132 transitions back to state 202, host busy, data available.

FIG. 12 is a state transition diagram for ECP controller 134. As with NIBBLE controller 132, the default state 220 for ECP controller is idled. Upon the start command, it passes into the setup state 222 and then into the forward idle state 224. States 224 and 226 are the primary data out states for Extended Communications Protocol controller 134, wherein Extended Communications Protocol controller 134 shifts between forward idle and sending bytes. If, during the course of sending data and shifting between states 224 and 226, controller 126 signals a TRY_REVERSE command. Extended Communications Protocol controller 134 will transition through the forward idle state 224 to the transition to reverse state 230.

From the transition to reverse state 230, the controller transitions to reverse idle state 232, and then alternately transfers back and forth between reverse idle 232 and the GET_BYTE 234 state, as data is transferred in reverse from the peripheral to the host.

If the reverse data is run length encoded, then Extended Communications Protocol controller 134 transitions from the GET_BYTE 234 state to run length encoded reverse idle state 238, from where it transitions to GET_BYTE state 240, and then back to reverse idle state 234 as the counted bytes are retrieved and sent.

While the preceding describes the automatic mode of operation under which parallel port interface module 20 is under automatic control of controller 136, as shown in FIG. 8, parallel port interface module 20 is capable of being transitioned into a non-automatic, register mode of operation, as shown in state 172 of state transition diagram FIG. 8. In this mode of operation, the parallel port interface module is operating under direct host control. In practice, it has been found to be a much slower mode of operation, but is necessary for use with peripheral devices which use additional commands and communications protocols which are not standard to the specifications for these protocols.

In register operation utilizing extended capabilities port language, ECP register mode controller 138 is utilized. A state transition diagram describing the operations of Extended Communications Protocol register mode controller 138 is shown and disclosed in FIG. 13. Its default state is idle. Upon transition of controller 126 to its register mode state 172 as shown in FIG. 8, and the receipt of a signal from the host to enable Extended Communications Protocol register mode controller 138, Extended Communications Protocol register mode operation controller 138 shifts from the idle state 250 into either the forward idle state 252 or reverse idle state 256, depending upon whether or not the command from the host is transitioned to either forward or not forward. If the transition is to the forward idle state 252, then the Extended Communications Protocol register mode controller 138 shifts between the forward idle state 252 of the send byte state 254 as long as output data is available, or until a command is received from the host to transition from the forward idle state 252 to the reverse idle state 256.

Upon transitioning to the reverse idle state 256, the Extended Communications Protocol register mode controller 138 will transition to GET_REVERSE byte state 258, and will alternately continue transitioning between reverse idle 256 and GET_REVERSE byte state 258 until the data is transferred.

If, however, the reverse data is run length encoded, then upon transitioning to the GET_REVERSE byte state 258, it will transition from there to run length encoded reverse idle state 260, and from there to get data bytes state 262, and then back to reverse idle state 256.

In register mode operations, parallel port interface module 20 is also capable of communicating with the peripheral using enhanced parallel port (EPP) protocols through enhanced parallel port register mode controller 136. FIG. 14 discloses a state transition diagram 4 and enhanced parallel port register mode controller 136. Again, in this mode of operation, the parallel port interface module is under direct host control through controller 136, and begins in its initial idle state 270. Upon receipt of an enhanced parallel port protocol signal from the host, the enhanced parallel port register mode controller transitions to an idle state 272, from where it can transition to either send byte state 274 or GET byte state 276.

While there is shown and described the present preferred embodiment of the invention, it is to be distinctly understood that this invention is not limited thereto but may be variously embodied to practice within the scope of the following claims.

What is claimed is:

1. A device connecting an external connection of a host device to an external connection of a peripheral device, comprising:

a universal serial bus port interface adapted to receive a serial bit stream of data from the host using a Universal Serial Bus communications protocol;

a controller adapted to extract data bytes from the serial bit stream of data and convert the extracted data bytes to comply with a IEEE 1284 communications protocol; and

a parallel port interface adapted to transmit the converted data bytes to the peripheral device using the IEEE 1284 communications protocol.

2. A device according to claim 1 including:

a memory storing a plurality of IEEE 1284 protocols in an hierarchical order; and

a circuit for converting data bytes received in the Universal Serial Bus communications protocol into data bytes in all of the IEEE 1284 communications protocols stored in said memory.

3. A device according to claim 2 including a logic circuit adapted to select one of the plurality of IEEE 1284 communications protocols for transmitting data bytes to said parallel port interface.

4. A device according to claim 1 including:

a memory storing IEEE 1284 communications protocol information and Universal Serial Bus communications protocol information; and

a circuit converting data bytes between the Universal Serial Bus communications protocol and the IEEE 1284 communications protocol according to the IEEE 1284 protocol information and the Universal Serial Bus communications protocol information stored in said memory.

5. The device of claim 1 including:

a serial register adapted to extract data from the serial bit stream from the host;

a circuit adapted to identify the data as command data, interrupt request data or payload data;

a circuit adapted to convert command data from the Universal Serial Bus communications protocol into command data in the IEEE 1284 communications protocol;

a circuit configured to convert events in the 1284 communications protocol into interrupt request data for the Universal Serial bus communications protocol; and

a circuit for converting payload data from the Universal Serial Bus communications protocol into payload data in the IEEE 1284 communications protocol.

6. A signal converter located between a host and a peripheral device, comprising:

a serial interface configured to receive from the host a serial bit stream using a serial bus communications protocol;

a circuit adapted to extract data from the serial bit stream received from the host;

a circuit adapted to convert the extracted data into a parallel data format; and

a parallel interface configured to transmit the data converted into the parallel data format to the peripheral device using a parallel bus communications protocol, wherein the serial bus communications protocol comprises a Universal Serial Bus protocol and the parallel bus communications protocol comprises a IEEE 1284 parallel bus protocol.

7. A signal converter located between a host and a peripheral device, comprising:

a serial interface configured to receive from the host a serial bit stream using a serial bus communications protocol;

a circuit adapted to extract data from the serial bit stream received from the host;

a circuit adapted to convert the extracted data into a parallel data format;

a parallel interface configured to transmit the data converted into the parallel data format to the peripheral device using a parallel bus communications protocol;

a memory storing specifications and information for the parallel bus communications protocol used by the peripheral device and the serial bus communications protocol used by the host device; and

a logic circuit adapted to use the stored specifications and information for converting data between the parallel bus communications protocol and the serial bus communications protocol.

8. A signal converter according to claim 7 including:

a logic circuit for determining different parallel bus communications protocols the peripheral device is capable of using;

a logic circuit adapted to select one of the parallel bus communications protocols; and

a logic circuit adapted to convert data bytes from the serial bus communications protocol to the selected one of the parallel bus communications protocols.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 6,218,969 B1
DATED : April 17, 2001
INVENTOR(S) : Watson et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Column 7,
Line 52, "nack" should read -- NACK --;

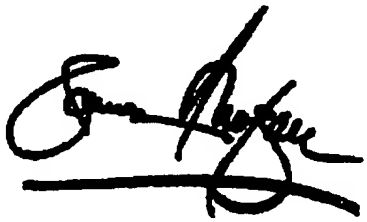
Column 8,
Lines 18 and 39, "nack" should read -- NACK --;
Line 50, "filter 144 5 is provided..." should read --filter 144 is provided ... --;
Line 65, "hierarchle" should read -- hierarchical --;

Column 9,
Line 1, "hierarchle" should read -- hierarchical --;
Lines 48-49, "interface module tries" should read -- interface module 20 tries --;

Column 11,
Line 52, "waits for if compatibility" should read -- waits for compatibility --.

Signed and Sealed this

Twenty-sixth Day of March, 2002



Attest:

Attesting Officer

JAMES E. ROGAN
Director of the United States Patent and Trademark Office

US006219736B1

(12) **United States Patent**
Klingman

(10) **Patent No.: US 6,219,736 B1**
(45) **Date of Patent: Apr. 17, 2001**

(54) **UNIVERSAL SERIAL BUS (USB) RAM ARCHITECTURE FOR USE WITH MICROCOMPUTERS VIA AN INTERFACE OPTIMIZED FOR INTEGRATED SERVICES DEVICE NETWORK (ISDN)**

(76) Inventor: **Edwin E. Klingman**, 3000 Hwy. 84, San Gregorio, CA (US) 94074

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: 09/191,443

(22) Filed: Nov. 12, 1998

Related U.S. Application Data

(63) Continuation-in-part of application No. 08/846,118, filed on Apr. 24, 1997, now Pat. No. 5,860,021.

(51) **Int. Cl.**[7] **G06F 13/00**

(52) **U.S. Cl.** 710/129; 710/14; 710/52; 710/100; 710/128; 709/226; 709/250; 370/259; 370/420; 370/524

(58) **Field of Search** 710/52, 14, 100, 710/128, 129; 709/250, 226; 370/259, 420, 524

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,604,683	8/1986	Russ et al.	710/100
4,799,156	1/1989	Shavit et al.	705/26

(List continued on next page.)

OTHER PUBLICATIONS

Don Johnson, "Universal Serial Bus System Architecture".

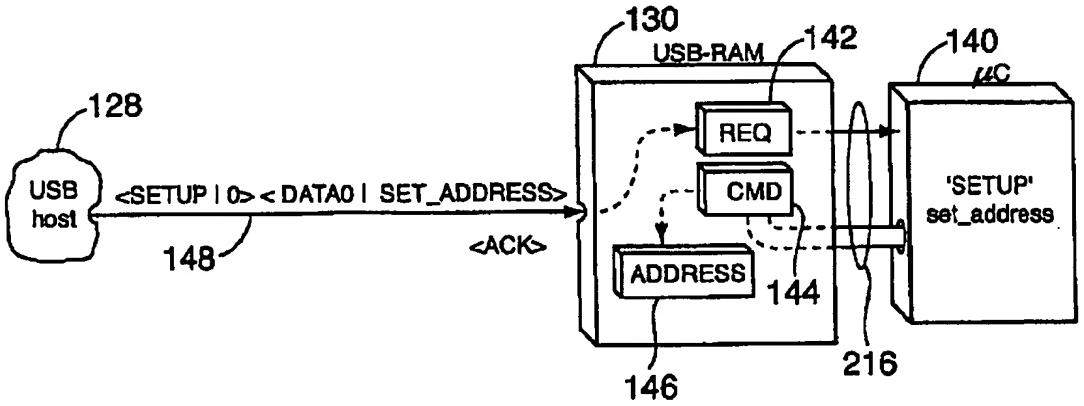
Primary Examiner—Thomas C. Lee
Assistant Examiner—Tammara Peyton

(74) *Attorney, Agent, or Firm*—Oppenheimer, Wolff & Donnelly, LLP; Claude A. S. Hamrick

(57) **ABSTRACT**

A RAM-based interrupt-driven interface device is disclosed for establishing a communication link between a universal serial bus (USB) host and a microcontroller device for providing a control function, the interface device being operative to receive digital information in the form of command, data and control packets from the host and to process the packets and communicate the processed digital information to the microcontroller device, and in response thereto, the microcontroller device being operative to communicate digital information to the interface device for processing and transfer thereof to the host. The interface device includes means for receiving a command generated by the host through a USB bus, means for storing the host-generated command and for generating an interface device interrupt signal upon storage of said host-generated command for use by the microcontroller device in responding to the host-generated command, a microcontroller bus for transferring microcontroller information and the interface device interrupt signal between the interface device and the microcontroller device. The interface device further includes means for receiving a microcontroller command from the microcontroller device in response to said interface device interrupt signal and means for storing the microcontroller command and it is operative to generate a microcontroller device interrupt signal upon storage of the microcontroller command for use by the interface device in developing an address for identification of the interface device to the host during subsequent communications therebetween, wherein during communication between the host and the interface device, the interface device-developed address is used by the interface device to identify host-provided information in the form of packets, and upon processing of the host-provided information, to provide the microcontroller device with the necessary information to allow it to respond to the host thereby allowing a generic microcontroller device to be flexibly interfaced with a USB, host for communication therebetween.

35 Claims, 10 Drawing Sheets

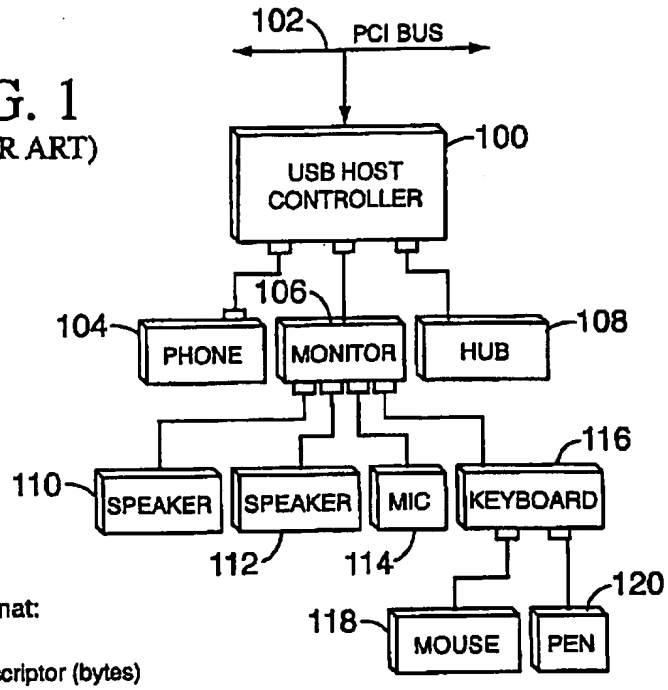


U.S. PATENT DOCUMENTS

5,309,440	*	5/1994	Nakamura et al.	370/420
5,497,373		3/1996	Hulen et al.	370/259
5,530,894		6/1996	Farrell et al.	709/250
5,537,654		7/1996	Bedingfield et al.	710/14
5,541,930		7/1996	Klingman	370/524
5,574,861		11/1996	Lorvig et al.	709/226
5,682,552		10/1997	Kuboki et al.	710/52
5,859,993	*	1/1999	Snyder	712/208
5,974,486	*	10/1999	Siddappa	710/53
6,044,428	*	3/2000	Rayabhari	710/129

* cited by examiner

FIG. 1
(PRIOR ART)



USB device descriptor format:

Length	size of the descriptor (bytes)
Type	uC device descriptor
bcdUSB	USB spec Release # in BCD
Class	Class code (assigned by USB)(if OxFF, vendor specific)
subClass	(assigned by USB)
Protocol	(assigned by USB)
MaxPkt	(8, 16, 32, or 64)
Vendor ID	(assigned by USB)
Product ID	(assigned by Manufacturer)
BCD Device	Device Release
iMFR	Index to string describing Manufacturer
iPROD	Index to string describing Product
iSER#	Index to string describing Serial #
numCONF	number of Configurations

FIG. 2
(PRIOR ART)

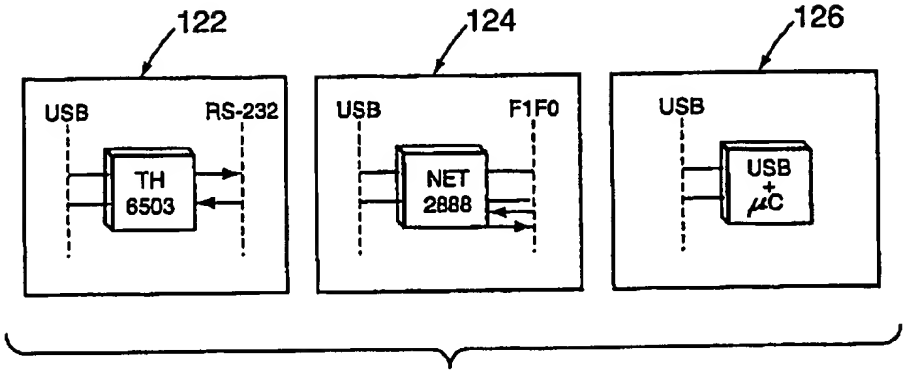

122
124
126
USB
RS-232
TH
6503
USB
F1F0
NET
2888
USB
USB
+
μC

FIG. 3

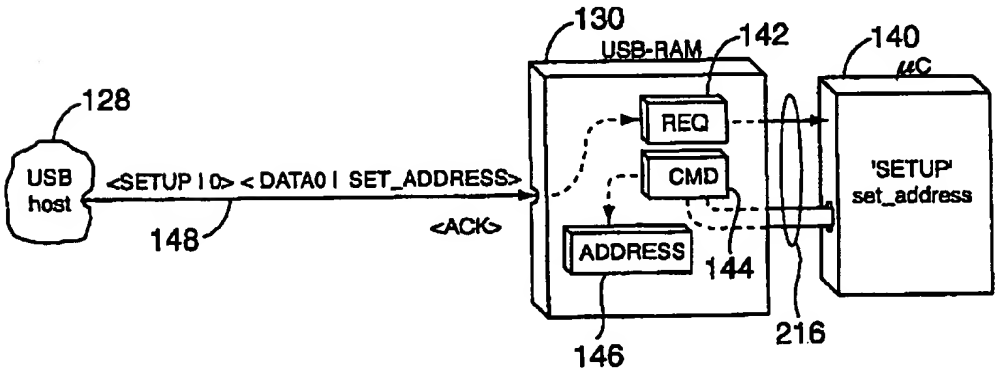

130
USB-RAM
142
140
μC
128
USB
host
<SETUP | 0> < DATA0 | SET_ADDRESS>
<ACK>
148
REQ
CMD
ADDRESS
144
146
216
'SETUP'
set_address

FIG. 4

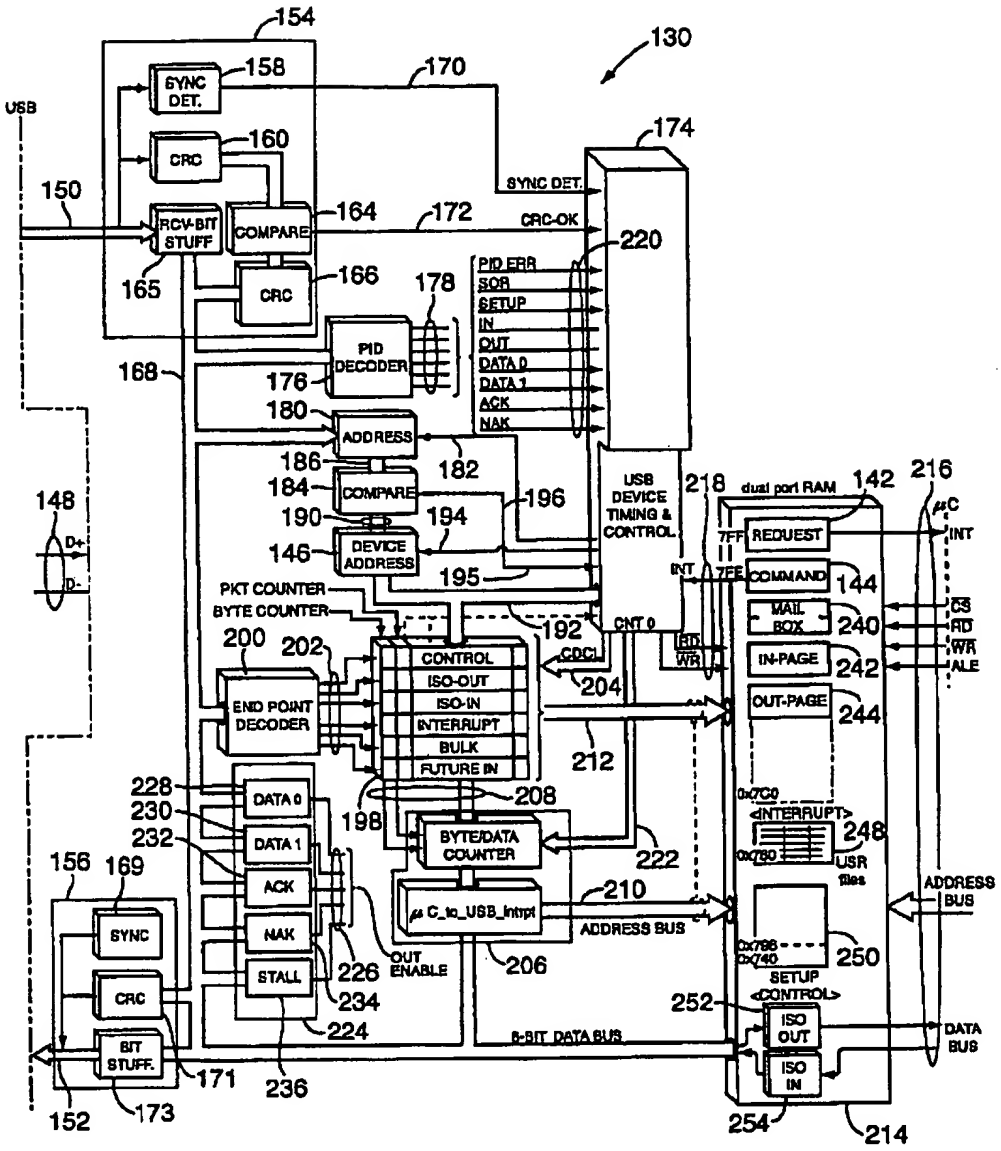

USB
150
154
158
SYNC DET.
160
CRC
RCV-BIT STUFF
165
COMPARE
164
166
CRC
168
170
172
CRC-OK
130
174
SYNC DET.
220
PID ERR
SOR
SETUP
IN
OUT
DATA 0
DATA 1
ACK
NAK
178
PID DECODER
176
180
ADDRESS
182
186
184
COMPARE
196
190
194
146
DEVICE ADDRESS
195
148
D+
D-
PKT COUNTER
BYTE COUNTER
192
200
202
END POINT DECODER
CONTROL
ISO-OUT
ISO-IN
INTERRUPT
BULK
FUTURE IN
208
198
204
212
USB DEVICE TIMING & CONTROL
218
dual port RAM
142
216
μC
INT
7FF
REQUEST
7FE
COMMAND
144
MAIL BOX
240
IN-PAGE
242
OUT-PAGE
244
CS
RD
WR
ALE
CNT 0
RD
WR
CDCI
228
230
232
DATA 0
DATA 1
ACK
NAK
STALL
226
234
224
236
156
169
SYNC
CRC
BIT STUFF.
171
152
173
BYTE/DATA COUNTER
222
μC_to_USB_Intpt
210
ADDRESS BUS
206
OUT ENABLE
8-BIT DATA BUS
0x07C0
<INTERRUPT>
248
0x0780
USR files
0x0796
0x0740
250
SETUP <CONTROL>
252
ISO OUT
ISO IN
254
214
ADDRESS BUS
DATA BUS

FIG. 5

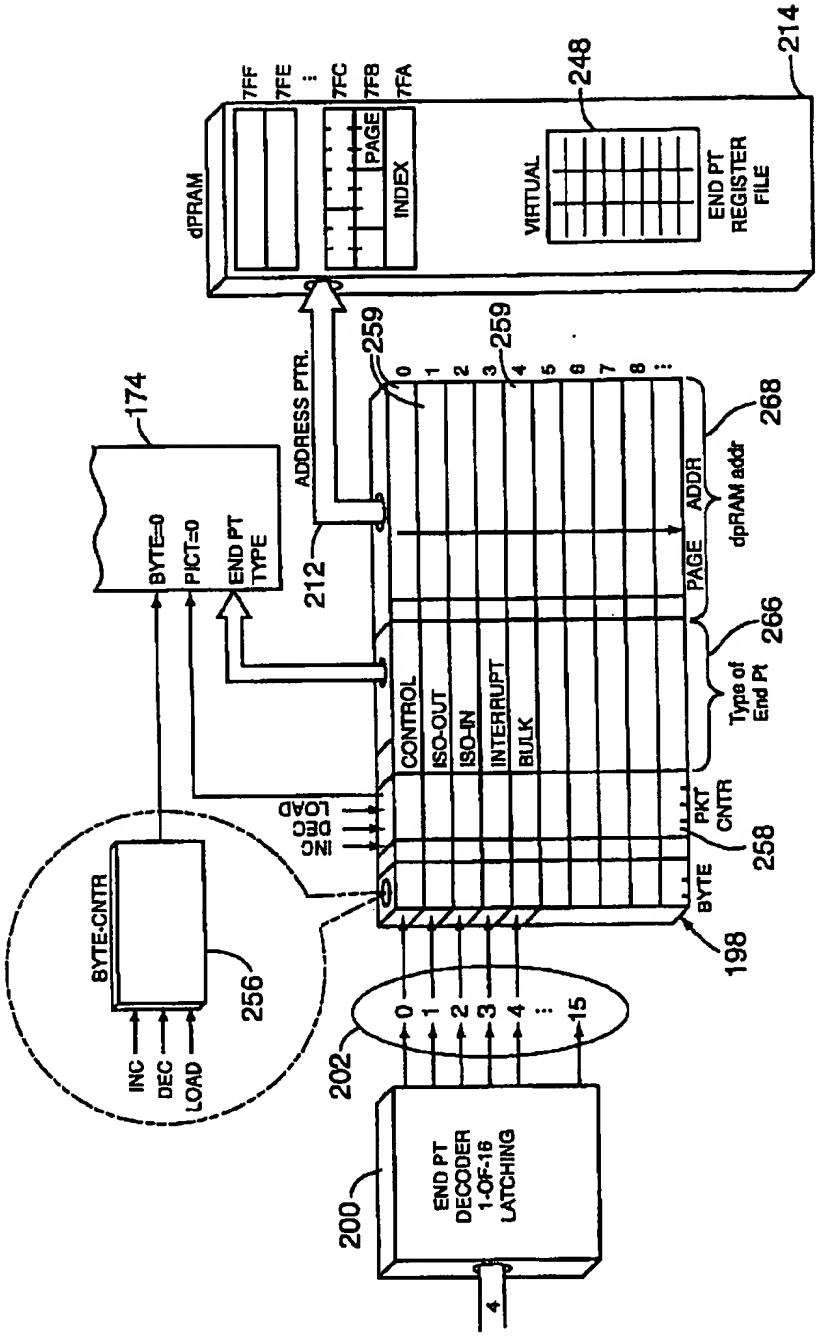
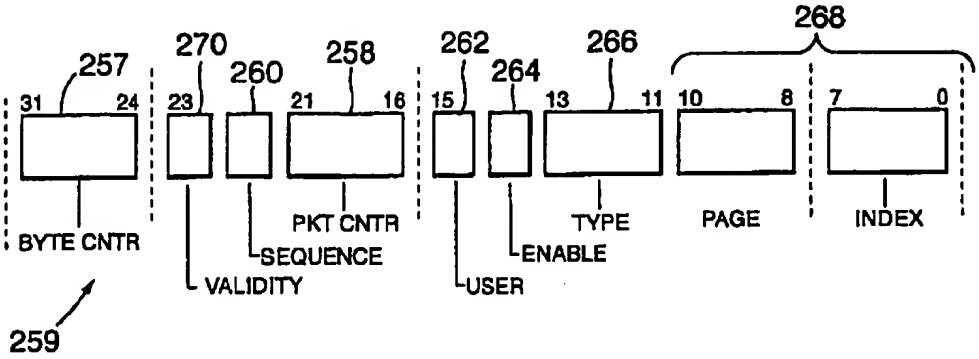

214
248
dPRAM
7FF
7FE
7FC
7FB
7FA
PAGE
INDEX
VIRTUAL
END PT REGISTER FILE
259
259
ADDRESS PTR.
174
BYTE=0
PICT=0
END PT TYPE
212
268
ADDR
dpRAM addr
PAGE
266
Type of End Pt
CONTROL
ISO-OUT
ISO-IN
INTERRUPT
BULK
INC
DEC
LOAD
PKT CNTR
258
BYTE
198
BYTE-CNTR
256
INC
DEC
LOAD
202
0
1
2
3
4
15
200
END PT DECODER 1-OF-16 LATCHING
4

FIG. 6



Field Name	Width (bits)	Position
Field Count	9	31..24
Validity Bit	1	23.
SEQ Bit	1	22
Packet Counter	5	21..16
User Bit	1	15
Enable Bit	1	14
EndPt Type	3	13..11
Page address	3	10..8
Page Index	8	7..0

FIG. 7

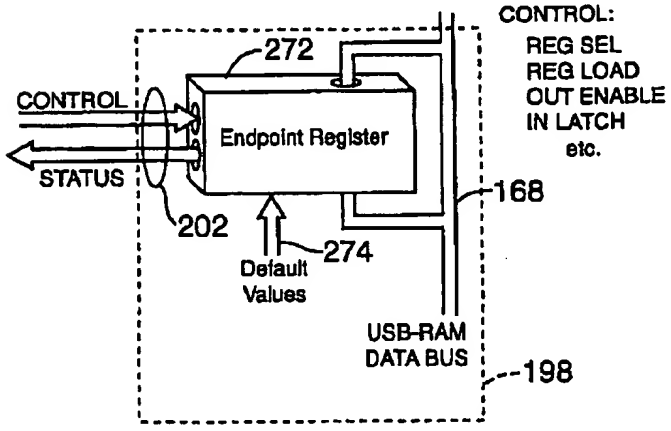


FIG. 8

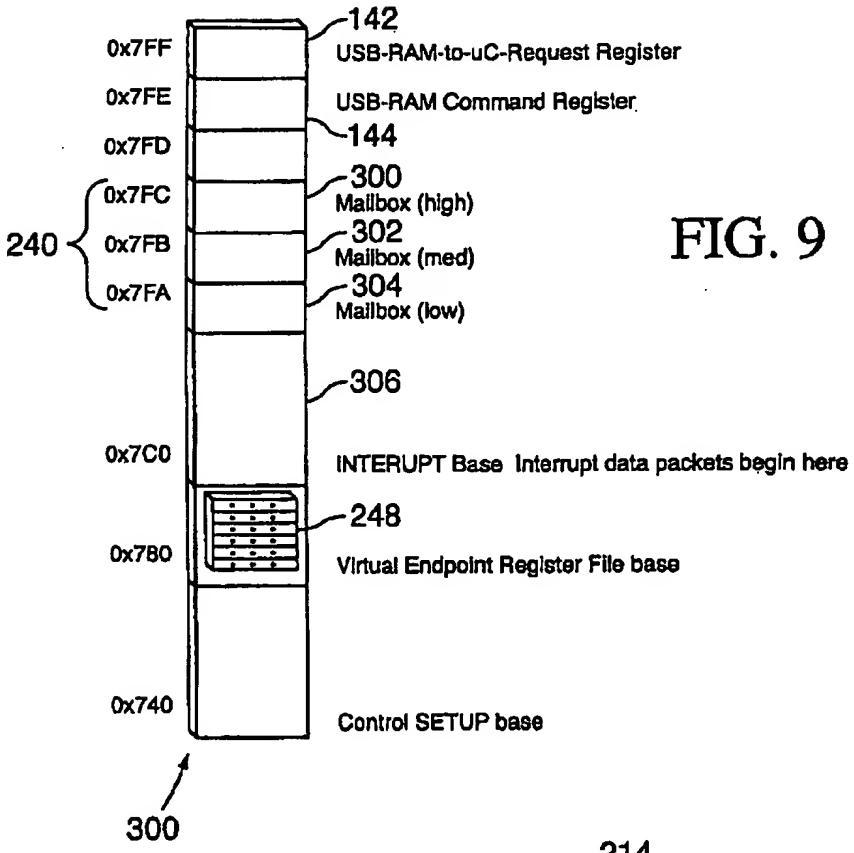

142
0x7FF
USB-RAM-to-uC-Request Register
0x7FE
USB-RAM Command Register
144
0x7FD
300
Mailbox (high)
0x7FC
240
0x7FB
302
Mailbox (med)
0x7FA
304
Mailbox (low)
306
0x7C0
INTERUPT Base Interrupt data packets begin here
248
0x780
Virtual Endpoint Register File base
0x740
Control SETUP base
300

FIG. 9

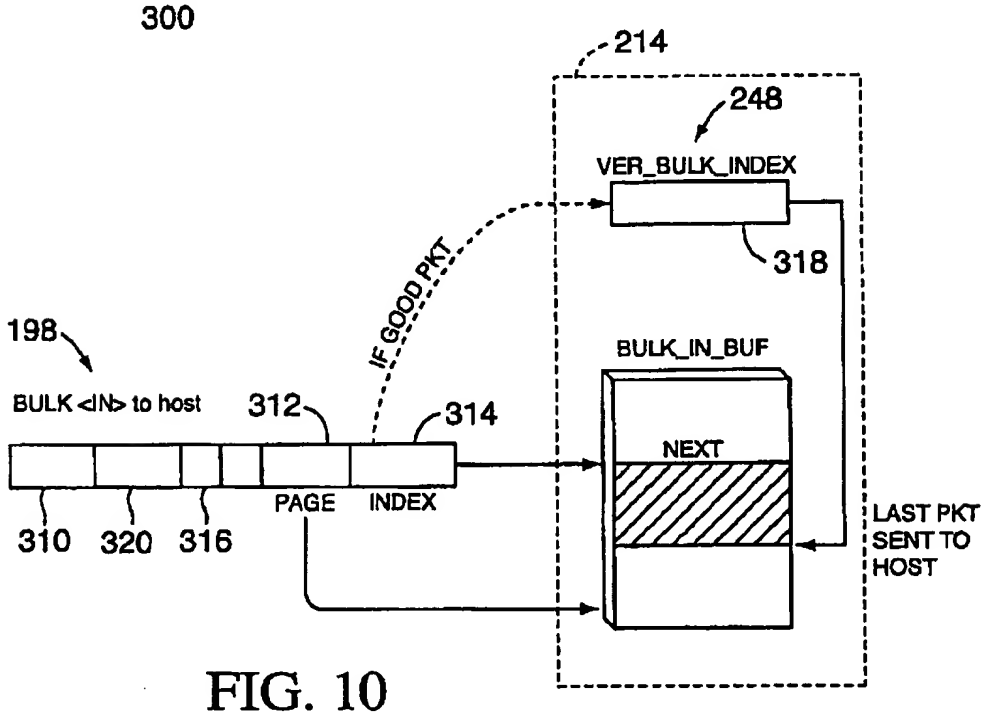

214
248
VER_BULK_INDEX
318
IF GOOD PKT
198
BULK_IN_BUF
BULK <IN> to host
312
314
NEXT
PAGE
INDEX
310
320
316
LAST PKT
SENT TO
HOST

FIG. 10

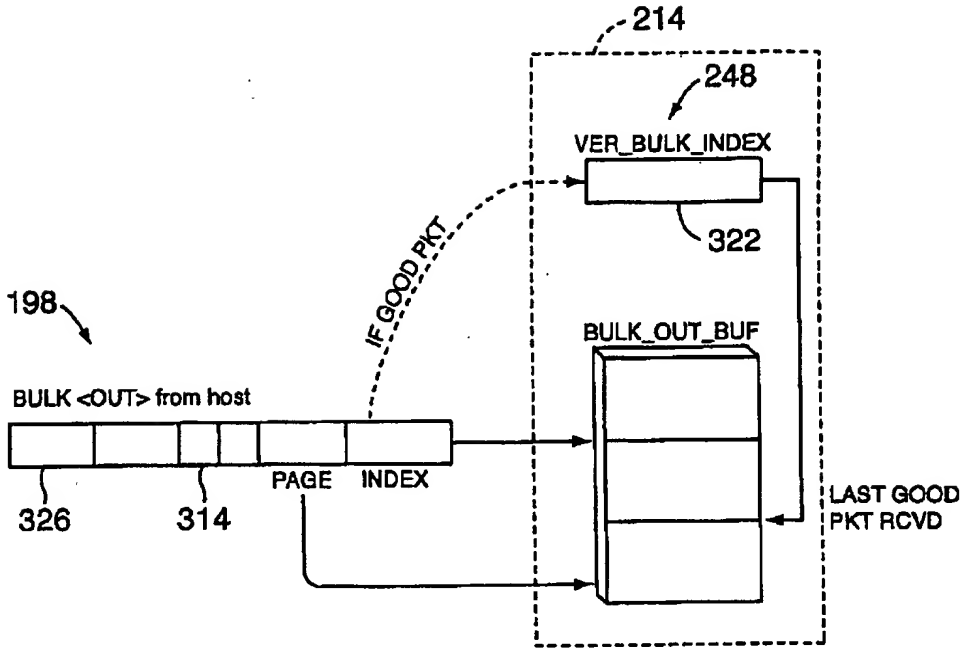

214
248
VER_BULK_INDEX
322
198
IF GOOD PKT
BULK_OUT_BUF
BULK <OUT> from host
PAGE
INDEX
326
314
LAST GOOD
PKT RCVD

FIG. 11

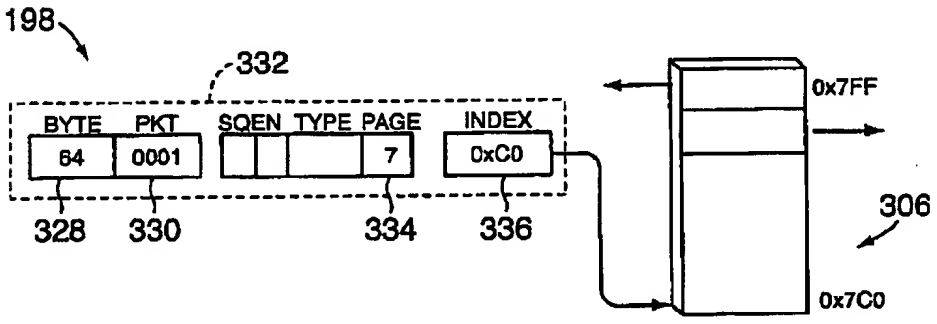

198
332
BYTE
PKT
SQEN
TYPE
PAGE
INDEX
64
0001
7
0xC0
328
330
334
336
0x7FF
306
0x7C0

FIG. 12

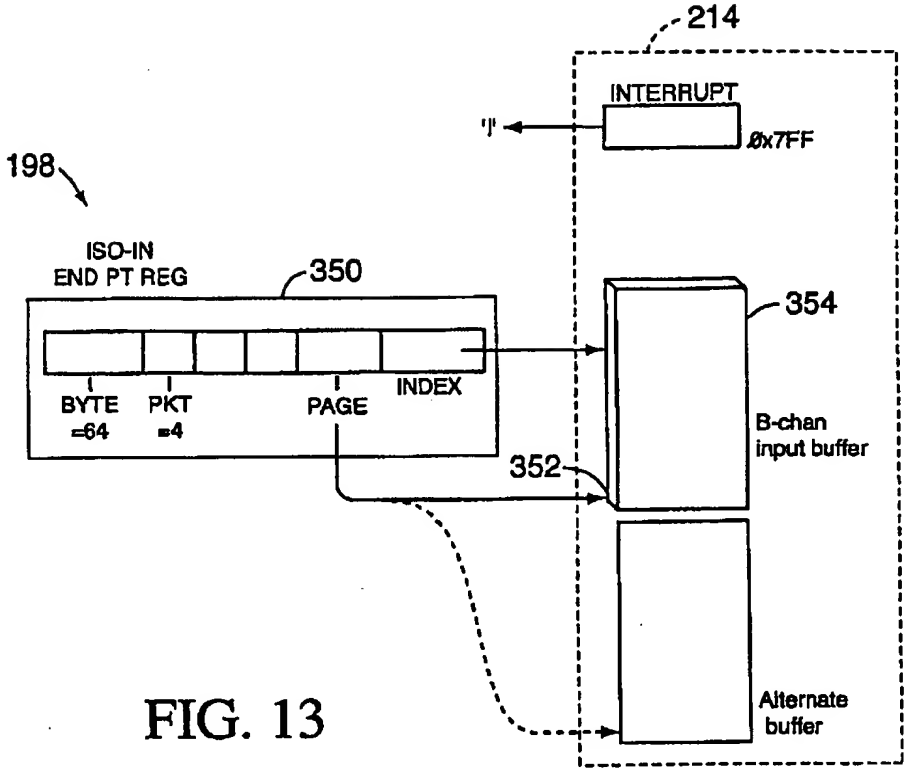

214
INTERRUPT
"I"
Øx7FF
198
ISO-IN
END PT REG
350
354
BYTE
=64
PKT
=4
PAGE
INDEX
B-chan
input buffer
352
Alternate
buffer

FIG. 13

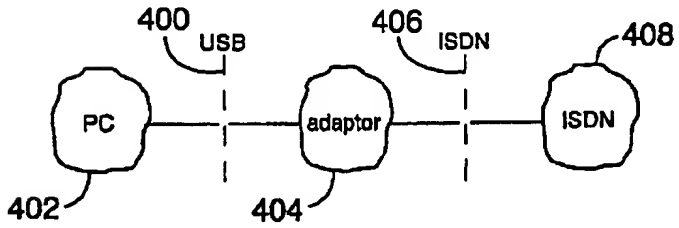

400
USB
406
ISDN
408
PC
adaptor
ISDN
402
404

FIG. 16

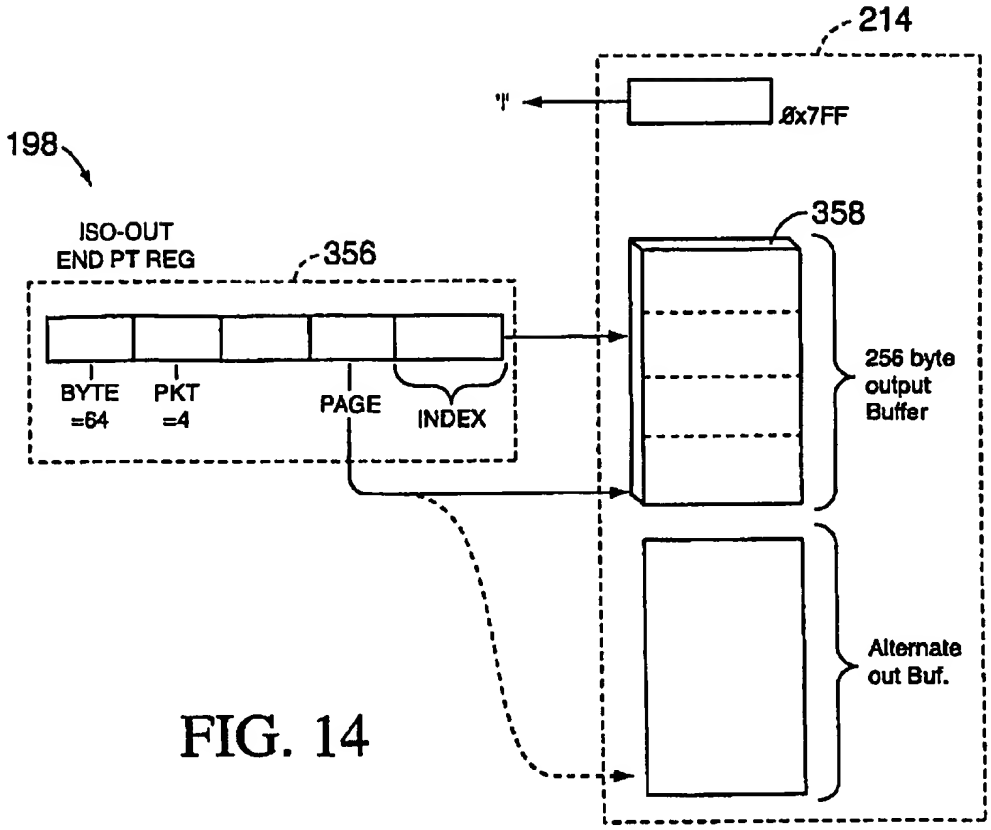

214
Ø
Ø x7FF
198
ISO-OUT
END PT REG
356
358
BYTE
=64
PKT
=4
PAGE
INDEX
256 byte
output
Buffer
Alternate
out Buf.

FIG. 14

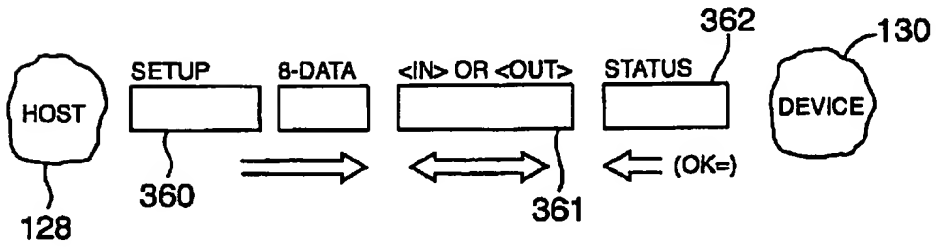

362
130
HOST
SETUP
8-DATA
<IN> OR <OUT>
STATUS
DEVICE
(OK=)
360
361
128

FIG. 15

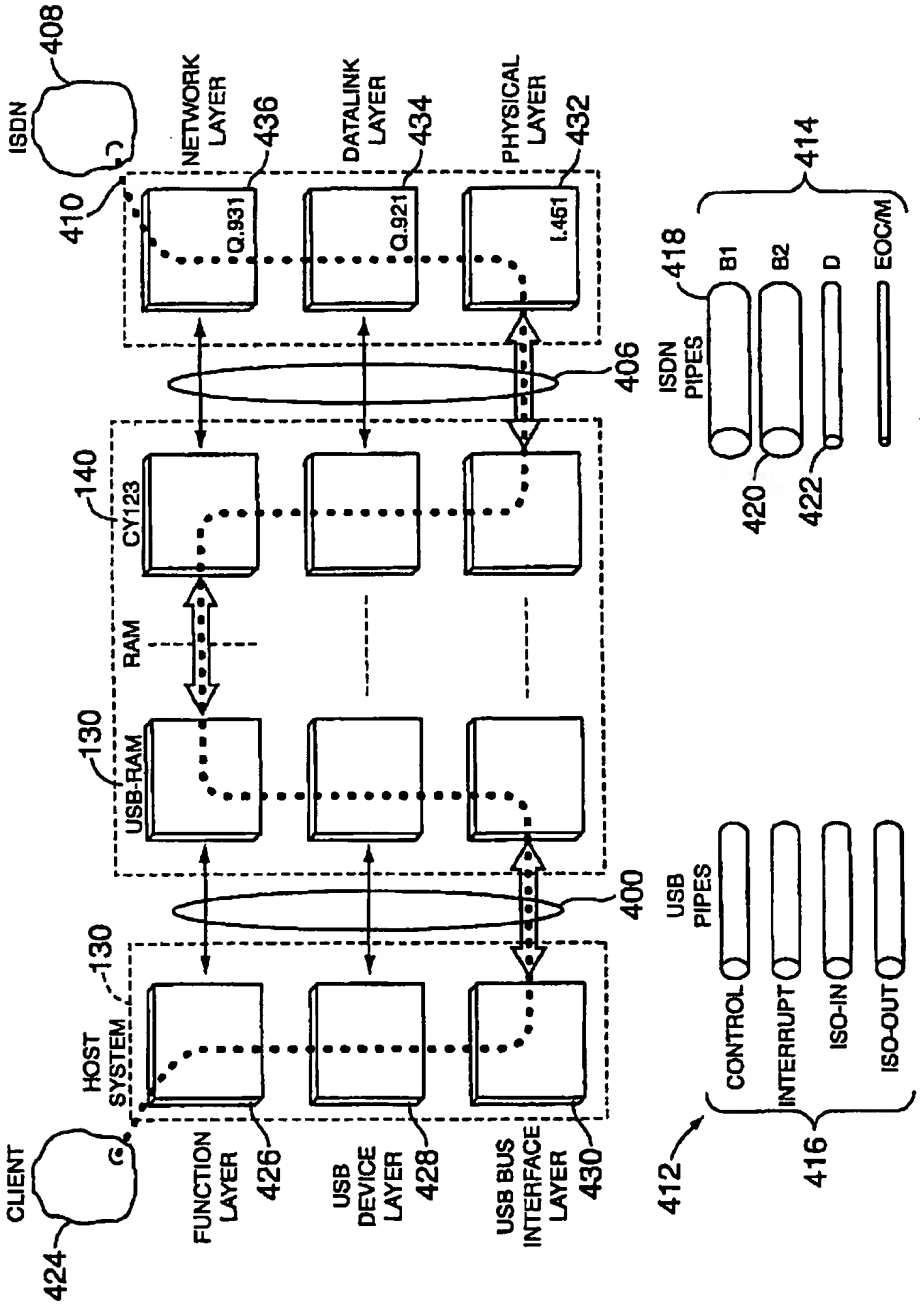

ISDN
408
410
NETWORK LAYER
436
DATALINK LAYER
434
PHYSICAL LAYER
432
Q.931
Q.921
I.451
406
140
CY123
RAM
130
USB-RAM
400
130
HOST SYSTEM
CLIENT
424
FUNCTION LAYER
426
USB DEVICE LAYER
428
USB BUS INTERFACE LAYER
430
414
418
B1
B2
D
EOC/M
ISDN PIPES
420
422
USB PIPES
CONTROL
INTERRUPT
ISO-IN
ISO-OUT
412
416

FIG. 17

UNIVERSAL SERIAL BUS (USB) RAM ARCHITECTURE FOR USE WITH MICROCOMPUTERS VIA AN INTERFACE OPTIMIZED FOR INTEGRATED SERVICES DEVICE NETWORK (ISDN)

CROSS REFERENCE TO RELATED APPLICATION

This application is a continuation-in-part of my prior application Ser. No. 08/846,118, filed Apr. 24, 1997, now U.S. Pat. No. 5,860,021, entitled "A SINGLE CHIP MICROCONTROLLER HAVING DOWN-LOADABLE MEMORY ORGANIZATION SUPPORTING "SHADOW" PERSONALITY, OPTIMIZED FOR BI-DIRECTIONAL DATA TRANSFERS OVER A COMMUNICATION CHANNEL."

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates generally to the field of general purpose microcomputers and particularly to a microcomputer unit including a serial interface controller such as the Universal Serial Bus (USB) RAM device to facilitate communication between a host and a microcontroller.

2. Description of the Prior Art

The growth of desktop computers has been accompanied by a corresponding growth in the number and types of peripheral devices that have various connection/interconnection schemes, etc. Accordingly, today's PC's have many peripheral connectors, most of which are expensive. As the size and cost of the PC decreases, the relative cost of these connectors increase. To alleviate this problem, high performance serial bus schemes are being defined that are designed to use one connector to attach many (lower performance) peripherals to the PC. Furthermore, due to the operational limitations of many of these peripheral devices with respect to what is referred to in the computer industry as "low speed", they typically require dedicated wires and connectors capable of supporting much higher speed data transfers than are required.

Moreover, information flows and the required responses over a high performance serial bus exceed the performance capability of generic microcontrollers of the type used in typical peripherals.

The Universal Serial Bus (USB) and "firewire" (IEEE 1394) has been introduced in the computer industry to effectuate "time sharing" of many of these low speed peripheral devices over a single higher speed connection thereby providing higher performance communication links while using such peripheral devices. This higher speed connection requires only minimal resources (such as I/O, DMA, Interrupt and Memory) from the host system. Prior art systems require such resources per peripheral.

By way of background, a summary of the USB and its operation is presented below. Although the preferred implementation of the serial interface bus is the USB, a similar approach will work with the faster "firewire" (IEEE 1394) operating at 100,200,400 . . . Mbits/sec.

DESCRIPTION OF THE UNIVERSAL SERIAL BUS

The characteristics of a USB communication link consists of a half duplex 12 Mbit/sec channel divided into 1.0000 millisecond "frames", which are distributed over a Tiered Star Topology.

FIG. 1 shows an example of a system using USB to communicate t a host (not shown). In FIG. 1, a USB host controller unit **100** is shown coupled to a PCI bus **102** for communicating information through the PCI bus to other peripheral devices, or hubs, that may be coupled to yet further peripheral devices. In FIG. 1, the peripheral devices: phone device **104**, a monitor device **106** and another hub device **108** are coupled through ports to the USB host controller device **100**. The monitor device **106** is further coupled to a plurality of other peripheral devices, such as two speaker units **110** and **112**, a microphone device **114** and a keyboard **116**. The keyboard **116** is further coupled to a mouse device **118** and a pen device **120** through ports.

All USB devices attach via a USB hub providing one or more ports. While each hub can provide either a high speed (12 Mb/s) or low speed (1.5 Mb/s) device support, only the high speed version will be considered for simplicity. Connectors and line characteristics are described in the USB Specifications, and are herein incorporated by reference. In the interest of maximum compatibility, Intel and the USB Implementors Forum make available a VHDL description of the Serial Interface Engine (SIE). A line driver (such as Phillips USB translation PDI-USB-P11) uses differential pair signaling with bit-stuffed NRZI (Non-Return-to-Zero, Inverted) coding.

Every transfer across a USB interface consists of a combination of packets. Four classes of transfers have been defined, each of which provides features useful to typical peripheral devices. Each transfer class will be described briefly:

Interrupt Transfer

Useful for devices that typically interrupt the host system in non-USB interface. USB interrupt transfers provide a maximum latency on the order of one millisecond, with average latency perhaps half that.

Control Transfer

Useful for sending specific requests from the host system to USB devices. This transfer is typically used during device initialization.

Bulk Transfer

Useful for data transfers that have no immediacy or periodicity requirements, such as the data returned from a floppy disk device.

Isochronous Transfer

Useful for periodic transfers or for devices requiring a constant data rate, such as voice communications over an ISDN phone.

A transfer class is typically associated with a device endpoint. The user of a USB device must analyze the transfer class(s) necessary for his purposes, and define appropriate endpoints. The endpoints are communicated to the USB host controller during the configuration process, using descriptors, which are data structures with a defined format. Each descriptor begins with a byte-wide field that contains the total number of bytes in the descriptor followed by a bytewide field that identifies the descriptor type. For endpoint descriptors, at least the following fields are required: Descriptor Length, Descriptor Endpoint Type, Endpoint address, and Endpoint attributes. An example of a device descriptor and the descriptor communications procedure is given in a following section.

The connection between client software on the host system and an endpoint on a peripheral device is via a Pipe. Typically a pipe connects the client data buffer on the host with an endpoint register on the device. The client software initiates a control transfer to read the device's descriptor(s), then registers the required endpoints with the system's USB host controller, which allocates USB bandwidth according to an implementation specific plan.

USB bandwidth allocation is highly flexible and device specific. Interrupt pipes can specify a latency ranging from one to 255 msec. An endpoint can define a maximum packet size, thereby allowing the host controller/allocator to compute the number of specific endpoints that can share a frame. Maximum packet size can be up to 64 for Interrupt endpoints, but as large as 1023 for Isochronous endpoints.

USB devices are not required to have a specific number or type of endpoint(s). The specific configuration for each device is set up during initialization. Since all SETUP and associated packets are CONTROL transfers, then at a minimum, any device must have at least one control endpoint. The USB-RAM interface described herein will support CONTROL, INTERRUPT, ISOCHRONOUS, and BULK transfers, as required by the microcomputer being interfaced to the USB-RAM.

CONTROL transfers begin with a setup stage containing an eight byte data packet, the eight bytes defining the type and amount of data to be transferred during the data stage. CONTROL transfers are guaranteed at least 10% bus allocation. In order to apportion control transfers over as many devices as possible, the data stage of a CONTROL transfer is limited to 64 bytes. Typical USB transactions consist of three phases:

Token Phase	Data Packet Phase	Handshake Packet Phase

All USB transactions begin with a token phase, defining the type of transaction to be broadcast over the USB. The four USB tokens are:

SOF (Start_of_Frame) begins each 1 ms frame

SETUP begins each CONTROL transfer

IN begins a data transfer from the device to the host

OUT begins a transaction to transfer data from the host to the device.

SOF and SETUP tokens are very specific, while IN tokens can be used in INTERRUPT transfers, BULK transfers, ISOCHRONOUS transfers, and the data phase of CONTROL transfers. The Token phase is always from the host to the device. The Data Packet direction varies according to the transaction, and the Handshake, if required, usually depends on the data direction. Each of the above packet phases transfers a packet with the following format:

[SYNC Seq.][Packet ID][Packet Info][CRC-bits][EOP]

The synchronizing sequence and End-of-Packet signal are handled by the Serial Interface Engine (SIE) and are not seen by the microcontroller, while packet bytes (exclusive of CRC bits) are handled by the microcomputer device in the present invention, thereby allowing maximum flexibility. In general, USB packets look like:

Packet ID
TRANSFER: [Sync] [SETUP] [Address] [Endpoint] [CRC-5] [EOP]
[OUT]
[IN]
SOF: [Sync] [SOF] [Frame #] [CRC-5] [EOP]
DATA: [Sync] [DATA] [Data payload] [CRC-16] [EOP]
HANDSHAKE: [Sync] [ACK] [EOP]
[NAK]
[STALL]

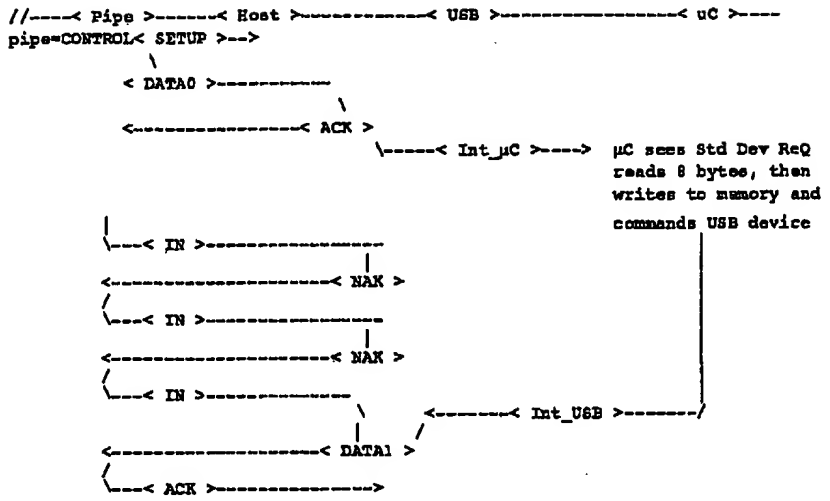
The USB-RAM enters a number of states in changing from 'unattached' to 'configured' (see below). Before being reset, the powered device will not respond to the bus. After reset, the USB-RAM responds to requests on its default pipe using either a unique assigned address or the default address.

USB Visible Device States
<Transition> State
not attached
<Attach>
Attached
<Power>
Powered
<Reset>
Default
<Address>
Addressed
<Configure>
Configured (Functional)
<Suspend>
Suspended

Suspended devices are maintained at a minimum power level, and are not functional. A USB-RAM exits the suspend mode when there is bus activity. The device may also request the host exit a suspend mode via electrical signaling to indicate a remote wakeup.

The normal sequence begins after reset with a host read on the default pipe to determine a maximum data payload available on the default channel, then the host assigns an address. The host reads the configuration information for each device configuration 0 to n then assigns a configuration value to the device, causing all endpoints to assume the characteristics of the configuration descriptor.

USB devices report their attributes to the USB client software using descriptors. The format of a USB device descriptor is shown in FIG. 2. USB protocols define several descriptors: DEVICE, CONFIGURATION, INTERFACE, ENDPOINT, STRING, and CLASS-specific. Each of these is requested via SETUP transactions in which the desired descriptor type is requested from the microcomputer. The preferred implementation uses the following procedure:



The general description of the USB scheme and prior art USB interfaces has been presented above. A time multiplexed medium speed serial bus is used to handle multiple low to medium speed devices, using multiple transfer types. At reset time the device must respond to packets on the default CONTROL pipe at address zero. Initial responses result in the device being assigned a unique address and in the device communicating descriptor information describing the number and types of endpoints that must be configured for the application that the device is serving. These and all following transfers are initiated by the USB host, typically via an IN or an out token, where the direction of information, i.e. IN and OUT, is relative to the host. The microcontroller then sends or receives a DATA packet as appropriate. Details of the present invention are described below.

As shown in FIG. 3, there are several approaches for establishing communications between peripheral devices and a host through the USB. One such approach at 122 is the USB-to-clocked serial interface, which uses the commercially-available Thesys TH6503 device. Another approach, shown in FIG. 3 at 124, is a USB-to-FIFO design using the NetChip NET-2888. A third approach, shown at 126, is to embed the USB device in a micro-controller, either an 8051 derivation such as the Intel 8x931, the Siemens C540U & C541U, the Anchor Chips EZ-USB, or the Cypress CY7C63001.

While the first approach, USB-to-clocked serial interface, is simple and useful for RS-232-like devices, it embodies all of the limitations of the RS-232-like serial devices. For example, RS232 on IBM PC's are (1) typically slow devices; (2) not well suited to multi-channel architectures; and (3) require considerable processor resources. The second approach is more useful for faster transfers, but typically requires DMA I/O to allow the controlling device to service the FIFO as required. The final approach, the USB embedded in a micro-controller, is well suited for bi-volume applications (Cypress details "mouse controller" application) but represents an extreme amount of design work (with minimal tools) for low to moderate volume applications.

The approaches presented above for interfacing with the USB have a number of shortcomings. One such shortcoming is that a very significant design effort is required, another is that these approaches are incompatible with a very large class of microcontrollers such as the Intel 8051, the Motorola 68xx, the Micron PIC, and similar 8-bit microcontrollers (also 4 and 16-bit), which typically do not include DMA circuitry, but do support memory interface and external interrupt(s).

The transfer of data between the high performance serial bus and a low performance generic microcontroller occurs via memory buffers that have specific locations and sizes. The locations and sizes will generally be specified by the microcontroller, and this information will be used during transfers, by the serial interface device. Because of the asynchronous relation between the serial bus and the microcontroller, arbitrating access to such buffer information is problematic.

Therefore, the need arises for an inexpensive device to interface peripheral devices, of various different types, such as currently-available microcontrollers, with a host through the USB or other bus devices while taking advantage of the high speed of the bus device.

SUMMARY OF THE INVENTION

Accordingly, it is an objection of the present invention to provide a Serial Interface Controller that uses buffering via a memory-based interface capable of generating interrupt signals to the generic microcontroller, and of coordinating data transfers between the host and the microcontroller, including flow control, and error handling and retry mechanisms.

The present invention represents a new architectural approach to solving the problems mentioned above. The invention provides a method and apparatus for providing a high performance serial interface between any commercially-available microcontroller device and the USB or other high performance serial bus. The architecture used in the presently preferred embodiment of the present invention (hereinafter referred to variously as the Serial Interface Ram (SI-RAM) or USB-RAM architecture) is related to the single chip processor unit design described in Applicants' pending U.S. patent application Ser. No. 08/846,118 filed Apr. 24, 1997 and entitled "A SINGLE CHIP MICROCONTROLLER HAVING DOWN-LOADABLE MEMORY ORGANIZATION SUPPORTING "SHADOW" PERSONALITY, OPTIMIZED FOR BI-DIRECTIONAL

DATA TRANSFERS OVER A COMMUNICATION CHANNEL". The application disclosure is expressly incorporated herein by reference.

An important advantage of the present invention is that it provides a high performance interface device for coupling a commercially-available microcontroller to the USB or other high performance serial bus for communication therebetween. The interface provides for rapid communication between the microcontroller and the serial bus device.

Another advantage of the present invention is that the interface appears to the microcontroller as a RAM device having interrupt capability thereby allowing any commercially-available microcontroller to interface with the USB.

Yet another advantage of the present invention is that it provides a general purpose USB-to-uC interface that is also optimal for interfacing to an ISDN adapter.

Briefly, a preferred embodiment of the present invention includes a RAM-based interrupt-driven interface device for establishing a communication link between a universal serial bus (USB) host and a microcontroller device for providing a control function, the interface device being operative to receive digital information in the form of command, data and control packets from the host and to process the packets and communicate the processed digital information to the microcontroller device, and in response thereto, the microcontroller device being operative to communicate digital information to the interface device for processing and transfer thereof to the host. The interface device includes means for receiving a command generated by the host through a USB bus, means for storing the host-generated command and for generating an interface device interrupt signal upon storage of said host-generated command for use by the microcontroller device in responding to the host-generated command, a microcontroller bus for transferring microcontroller information and the interface device interrupt signal between the interface device and the microcontroller device. The interface device further includes means for receiving a microcontroller command from the microcontroller device in response to said interface device interrupt signal and means for storing the microcontroller command and it is operative to generate a microcontroller device interrupt signal upon storage of the microcontroller command for use by the interface device in developing an address for identification of the interface device to the host during subsequent communications therebetween, wherein during communication between the host and the interface device, the interface device-developed address is used by the interface device to identify host-provided information in the form of packets, and upon processing of the host-provided information, to provide the microcontroller device with the necessary information to allow it to respond to the host thereby allowing a generic microcontroller device to be flexibly interfaced with a USB host for communication therebetween.

These and other advantages of the present invention will no doubt become apparent to those skilled in the art after having read the following disclosure which makes reference to the several figures of the drawing.

IN THE DRAWINGS

FIG. 1 is a diagram illustrating an example of a system using USB to communicate to a host.

FIG. 2 presents the format of a USB device descriptor.

FIG. 3 shows several approaches for establishing communications between peripheral devices and a host through the USB.

FIG. 4 illustrates a preferred embodiment of the present invention to include a USB host coupled, through a communication link, to a USB RAM device 130

FIG. 5 shows a detailed view of the internal architecture of the USB RAM device shown in FIG. 4.

FIG. 6 shows a more detailed schematic of the substructure in FIG. 5.

FIG. 7 shows the organization of an endpoint register within the endpoint register file 198 of FIG. 5.

FIG. 8 shows interfacing to an endpoint register.

FIG. 9 shows a memory map, which is the default memory map (or organization) of information stored in the dual port RAM device.

FIG. 10 shows a Bulk-IN endpoint register storage location is shown included within the endpoint register file.

FIG. 11 shows a Bulk-Out endpoint register storage location is shown included within the endpoint register file.

FIG. 12 shows the default INTERRUPT endpoint register of the endpoint register file.

FIG. 13 shows the configuration of the default ISO-IN endpoint register of the endpoint register file.

FIG. 14 shows the configuration of the ISO-OUT end point register 356.

FIG. 15 shows a conceptual representation of control transfers.

FIG. 16 illustrates a high level diagram including two major interfaces: a USB interface and an ISDN interface.

FIG. 17 shows the layered architecture of ISDN as supported by the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

In FIG. 4, a preferred embodiment of the present invention is shown to include a USB host 128 coupled, through a communication link 148, to a USB RAM device 130. The USB RAM device 130 is further coupled to a microcontroller device 140 via microcontroller lines 216. The USB RAM device 130 includes a request storage location 142, a command storage location 144 and an address storage location 146. As will be further described below, storage locations 142 and 144 reside within a random-access-memory (RAM) device while the storage location 146 is included in a register.

The request storage location 142 operates to store commands provided by the USB host 128 for use by the microcontroller device 140. The command storage location 144 operates to store a microcontroller-provided command, which ultimately provides an address in the address storage location 146 of the USB RAM device 130. The USB RAM device 130 is assigned an address by the USB host 128. This process is performed by the USB host 128 initiating an address configuration procedure. During such a configuration process, the USB RAM device 130 is assigned a unique address that it uses for detecting its identity among other USB devices which may also be coupled to communicate with the USB host.

Resetting of the USB RAM device 130 will invoke the device to respond to a default address of zero. It should be noted that each device that is coupled to the USB host 128, other than the USB RAM 130, is also assigned a unique address prior to transfer of any information.

A "SET_ADDRESS" request is used to assign the USB RAM device 130 its unique address. The microcontroller device 140 is responsible for interpreting the "SET_

ADDRESS", request which is transparent to the USB RAM device **130**. The USB RAM device **130** detects a "SETUP PID" (Packet Identification), i.e. SETUP|0 with PID being the value '0', command, and signals the microcontroller device **140** when the device request has been received and stored via the request storage location **142** from the USB host **128**. The microcontroller device **140** determines that the request stored within the request storage location **142** signals the device address assignment by decoding the "SETUP PID" and thereafter writes a SET_ADDRESS command to the USB RAM device **130** by storing the command in the command storage location **144** and storing the address at a specified location in RAM. The USB RAM device **130** then copies the address information into the address storage location **146**.

FIG. **4** generally illustrates the basic communication protocol between the USB RAM device **130** and the microcontroller **140** when a SETUP command is initiated by the USB host **128**. The internal architecture within the USB RAM device **130** is then used to facilitate the command protocol as will be described in further detail below.

It should be further noted that the communications protocol between the USB RAM device **130** and the USB host **128** is governed by the USB specification, which is a known standard in the industry and is described in a publication entitled "Universal Serial Bus System Architecture" by Don Anderson. Data that is communicated via the communication link **148** between the USB host and the USB RAM device is performed in a serial fashion in the form of packets.

FIG. **5** is presented to show a detailed view of the internal architecture of the USB RAM device **130**. In FIG. **5**, the communication link **148** is shown as a full duplex communication link having D+ and D– lines coupled to transfer information between the USB host **128** (not shown) and a receiver line **150** and transmittal line **152**. That is, the communication link **148** couples information received from the USB host serially onto the receiver line **150** and similarly transfers information from the serial transmitter line **152** through the communication link **148** to the USB host.

The receiver line **150** is shown coupled to a serial interface engine receiver **154** and the transmittal line **152** is shown coupled to a serial interface engine transmitter **156**. The design and VHDL specifications for the serial interface engine receiver **154** and the serial interface engine transmitter **156** are commercially made available to users by Intel Corporation.

The serial interface engine receiver **154** operates to convert received serial data from a nonreturn-to-zero (NRZ) format to a binary format for use by the USB RAM device **130** and the serial interface engine transmitter **156** similarly converts serial binary data to NRZ, data for communication to the USB host.

Included in the serial interface receiver **154** is a receiver sync detector circuit **158** coupled to receive serial information from the receiver line **150** and operates to generate a sync detect signal that is coupled onto a sync detector line **170**. The serial interface engine receiver **154** further includes a receiver CRC circuit **160** coupled to the receive line **150**. Further included within the serial interface engine receiver **154** is a receiver bit stuffing circuit **165** coupled to the receiver line **150** for removing bits that are included in a received packet that are neither sync nor CRC bits and provide no valuable information to warrant decoding thereof. The receiver bit stuffing circuit **165** is connected to the an internal data bus **168** and it is further coupled to a receiver CRC generator circuit **166**. The circuit **166** is coupled to receive information from a receiver comparator circuit **164** which is in turn coupled to receive information from the receiver CRC circuit **160**.

As described earlier in this document, typical USB transactions consist of three phases: a token phase; a data packet phase; and a handshake packet phase. Each of these packet phases is arranged in a given format having a sync, a packet ID, packet information, CRC information, and EOL information (the latter for identifying when the communications line is going to be idle). Accordingly, when information is received through the communication link **148**, the sync portion of the packet is detected by circuit **158** and the CRC portion of the packet is detected by the circuit **160**. The CRC portion of the packet is compared, using the circuit **164**, to a generated CRC, which is developed by the circuit **166**. The outcome of this comparison is a generated signal, by the circuit **164** and coupled onto a CRC OK line **172** such that when the two CRC values match, the signal that is coupled onto the CRC OK line **172** is activated. The sync detector line **170** and the CRC OK line **172** are further used to provide coupling between the serial interface engine receiver **154** to and a USB-RAM timing and control circuit **174**. The detailed design of the timing and control circuit **174** is described in the form of 'pseudo-code' in Appendix A attached hereto.

During transmission of data from the USB RAM device to the USB host, the serial interface engine transmitter transfers serial data via the transmitter line **152**. The serial interface engine transmitter **156** includes a transmitter sync circuit **169** coupled to the transmitter line **152** for developing the sync portion of a packet. The serial interface engine transmitter **156** further includes a transmitter CRC circuit **171** coupled to receive data from the internal data bus **168** and coupled to generate a CRC bit pattern onto the transmitter line **152**. The serial interface engine transmitter further includes a transmitter bit stuffing circuit **173** which is coupled to the internal data bus **168** and further coupled to the transmitter line **152**.

A Packet IDentification (PID) decoder circuit **176** is connected to the internal data bus **168** for decoding packet identification information from each formatted packet received and accordingly generates control signals that are coupled onto a PID control bus **178** for use by components of the USB RAM device.

In FIG **5**, additionally shown, is an address register **180** that is coupled to the internal data bus **168** for receiving address (or identification) information from the USB host. The address register **180** stores the received address information and couples the same onto an address bus **186** for use by an address comparator circuit **184**. The address register **180** operated to store a new address upon activation of a signal that is coupled onto an address latch line **182** by the circuit **174**. The address comparator circuit **184** compares the address information that is stored in the address register **180** to the address information that is stored in the address storage location **146** and generates a signal in response thereto that is coupled onto an address match line **195**.

The address storage location **146** is further coupled to an endpoint register file **198** and to the circuit **174** through a timing and control address bus **192**. The address latch line **182** and the address match line **195** are connected to the circuit **174**. The device address latch line **194** is also connected to the circuit **174**. The circuit **174** is further connected to a timing and control bus **220** and generates signals coupled onto two busses: the endpoint register file control bus **204**; and the working pointer control bus **222**.

The circuit 174 further generates and receives signals through a dual port RAM control bus 218. The circuit 174 consists of hardware components and executes a program for generally arbitrating the flow f information among the remaining components within the USB RAM device 130. In so doing, the circuit 174 generates and receives control information used to direct information traffic through the device 130.

The endpoint register file 198 is coupled to receive information from an endpoint control bus 202. The endpoint control bus 202 communicates information from an endpoint decoder circuit 200 which is in turn coupled to the internal data bus 168. The endpoint register file 198 is operative to generate information through a working pointer bus 208 to a working pointer circuit 206. The working pointer 206 is further coupled to the internal data bus 168. The endpoint register file 198 is comprised of endpoint registers with each register for storing information that pertains to an endpoint. The contents of the endpoint register file 198 will be further explained below.

The working pointer circuit 206 is coupled to a dual port RAM device 214 through an internal address bus 210 that is generated by the working pointer circuit 206. The dual port RAM device 214 is coupled to: circuit 174 through the bus 218; endpoint register file 198 through an endpoint address pointer 212; and internal data bus 168. The dual port RAM device 214 is further coupled the microcontroller device 140 through microcontroller lines 216. The microcontroller lines 216 include an address bus, a data bus and control lines, the latter for coupling chip select, read, write, ALE and INT signals therethrough.

Data is transferred between the USB RAM device 130 and the microcontroller device 140 through the bi-directional data bus of the lines 216. The read and write signals coupled onto the lines 216 identify the direction of data flow between the USB RAM device 130 and the microcontroller device 140. The ALE signal is used for accommodating 8 and 16-bit addressing schemes. For example, if the address bus included within lines 216 is 16 bits wide, the 8 most significant bits of the address lines are first captured in a latch or register device (not shown) using the ALE signal by the USB RAM device 130 before arrival of the 8 least significant bits at which time, both portions of the address are concatenated to form a 16 bit address information for use in reading and writing data in the USB RAM device 130.

The dual port RAM device 214 is a sophisticated storage device having an associated memory map that is particularly suited for USB applications. The memory map associated with the dual port RAM device 214 includes the request storage location 142 which is mapped to the top of the memory at a location identified by '0x7FF' (in hexadecimal notation). When the request storage location 142 is addressed and written to, an interrupt is generated to the microcontroller device 140 through the INT signal that is coupled onto the lines 216.

The dual port RAM device 214 further includes the command storage location 144 which is mapped to address location 0x7FE. When the command storage location 144 is written to by the microcontroller device 140, an interrupt is generated and received by the circuit 174 through the interrupt line of the control bus 218. The dual port RAM device 214 further includes a mailbox storage location 240 and IN-PAGE storage location 242 and an OUT-PAGE storage 244. Further included within the device 214 is an area for storing virtual endpoint register information at a virtual endpoint register file storage location 248 which is mapped from address locations 0x7C0 to 0x780. The device 214 further includes a SETUP storage location 250 for storing the SETUP control information as discussed earlier. The dual port RAM device 214 further includes an ISO OUT data storage location 252 and an ISO IN data storage location 254. The location 252 is used to store data when data is being transferred from the USB host to the microcontroller device 140 and the location 254 is used to store data information when data is transferred from the microcontroller 140 to the USB host 128.

The internal data bus 168 is further coupled to a transmitter information generator circuit 224, which is coupled to the circuit 174 through transmitter generator control bus 226. The circuit 224 includes a DATA0 generator circuit 228 which is coupled to output to the internal data bus 168 and further coupled to the circuit 174 through the bus 226. The circuit 224 further includes a DATA1 generator circuit 230, an ACK generator circuit 232, a NAK generator circuit 234, and a STALL generator circuit 236 which are all coupled to the internal data bus 168 and further coupled to the circuit 174 through the bus 226. Information from the circuit 224 is received through the internal data bus 168 by the transmitter 156 and converted to NRZ format for transmission to the USB host.

Pursuant to the USB serial protocol, data transferred via serial packets is either read from or written to the USB RAM device 130. As will be apparent further below, when written to, the two highest RAM locations in the dual port RAM device 214 cause respective interrupt signals to be asserted and when read from the RAM locations, cause the respective interrupt signals to be cleared. Access to these locations, and, typically, access to associated data in RAM requires the use of a working pointer on either side of the dual port RAM, i.e., the address bus 212 on one side and the address portion of the bus 216 on the microcontroller side. The preferred implementation utilizes a working pointer per end point in 198, although by the nature of the time shared bus, only one pointer is active at a time. In addition, a separate pointer 206 is dedicated to service interrupts.

Handshake responses to USB packets must occur in approximately one microsecond. Even with interrupts, this is considerably faster than typical microcomputers can respond, therefore the architecture must compensate for this mismatch. The preferred compensation implementation employ the use of 'auto-NAK' wherever feasible, and the use of "pre-configuration." For example, only one pipe at a time can be active, therefore, in principle one working pointer will suffice to read and write packets in the USB RAM 130. However, if the microcomputer cannot respond fast enough to setup the pointer for each pipe, then the pointer would have to point to the same default memory location for all pipes. But this then requires the microcomputer to load and unload each data packet quickly and to free up the memory for the next transaction. In general, this is not practical, therefore we use the "pre-configure" strategy: each endpoint has an associated virtual endpoint register, within the virtual endpoint register file storage location 248 of the dual port RAM device 214, identified within the device 214 by a pre-assigned address per each endpoint register. Using this address, each endpoint register may be pre-loaded during reset by the microcontroller device 140.

Because different endpoints may have different maximum packet sizes associated with them, it is convenient to associate these values with the pointer registers, so that, at the same time the working pointer in 198 is loaded with a specific endpoint, the corresponding counter is also loaded:

Thus the microcontroller device **140** can preload the counters and address registers of the virtual endpoint register file **248** that is associated with each endpoint with unique values when the USB RAM device **130** is in reset. These default values can be chosen to optimize, in some sense, the distribution of end point buffers over the dual port RAM device **214**. While optimal for a "typical" system, such values are almost never ideal for various particular systems. Thus, the programmability of the registers and counters associated with endpoints, allows for the 'best' distribution of endpoint buffers for any given application thereby maximizing flexibility.

Each of the circuits **232**, **234** and **236** respectively generate the ACK, NAK and STALL handshake packets discussed earlier. With each of these, a sync pattern is appended to the packet information before transmission thereof to the USB host (with no CRC being necessary to transmit). However, an EOP which basically has the effect of causing the communication line to become idle is transmitted. The circuits **228** and **230** operate to couple data information onto the internal data bus **168** for transmission thereof through the circuit transmitter **156** to the USB host.

The endpoint decoder circuit **200** is a 1 to 16 decoder and the endpoint register file **198** which works in combination with the endpoint decoder **200** is not simply a register file, but rather a sophisticated device for storing endpoint register information in an elaborate fashion, as will be described further below.

The PID decoder circuit **176** as described above, decodes the packet ID information and based upon the information included within the packet ID generates control signals to the circuit **174** via the bus **178**. The packet ID, and may be any one of SETUP, OUT, IN, SOF, DATA, ACK, NAK information and based upon such information control lines such as delineated within bus **220** generated by the packet decoder circuit **176**. Further generated by the packet decoder circuit **176** is a PID error signal coupled through the bus **178** to the bus **220** for receipt by the circuit **174**. This signal is used to detect a packet identification information has been received in error. The internal data bus **168** is 8 bits wide, while the internal address bus(es) **210(212)** are at least 11 bits wide. As each endpoint is communicated from the USB host to the USB RAM device **130** using descriptors, the field of the descriptor is used to select the appropriate endpoint register within the endpoint register file **198**. Each of the registers within the register file **198** serves an endpoint such as CONTROIL, ISO-OUT, ISO-IN, INTERRUPT, BULK. Each field of the descriptor stored within the register file **198** has a format as shown in FIG. **7**.

In FIG. **6**, a more detailed schematic is shown of the structure in FIG. **5** that handles most of the endpoint information. The endpoint register file **198** is shown to include rows of storage locations, each row for storing an endpoint register **259**. Each endpoint register **259** is 32 (=0 . . . 31) bits wide and these bits are grouped into fields as will be described further with respect to FIG. **7**. A byte counter **256** is shown included within the byte counter field of each of the endpoint registers **259** although as indicated earlier, only one byte counter is necessary for all endpoints. The type of ENDPT field **266** tells **174** what type of endpoint is being serviced.

Each row of the register file **198** in FIG. **5** contains 32 bits as shown by a representative row in FIG. **7**. In FIG. **7**, starting from the least significant bit, 8 bits (0 . . . 7) are designated for the storage of the index information. The next 3 bits, bits **8** through **10** identify the page number. The following 3 bits identify which endpoint type is being stored, i.e., control ISO-IN, ISO-OUT, etc. The next bit distinguishes between a NACK and an ACK packet and the next 5 bits are the packet count; the next bit **22**, identifies whether there is a DATA 1 versus DATA **0** type of information. The next bit after that, which is bit **23** is dedicated to validity check and the next 8 bits. bits **24** through **31** are byte counter bits.

Each of the 32-bit endpoint registers **259** has an image in the virtual storage location **248** within the dual port RAM **214** shown in FIG. **5**. The virtual storage location **248** holds the endpoint register information and this information is loaded into the endpoint register file **198** when the USB RAM device **130** is configured as discussed above. Each field is thereafter selectively re-loaded, as appropriate. Each of the fields of an endpoint register in FIG. **7** is further described below. Byte Counter

An 8 bit, bits **24** through **31**, endpoint byte counter field **257**, shown in FIG. **7**, is used to preload the endpoint byte counter **256** (shown in FIG. **6**). The nature of the USB packet protocol ensures that only one packet will be on the bus at any given time and packets are not pre-emptiable. Therefore, the device requires one byte counter regardless of the number of endpoint registers implemented.

When an IN token on a specified endpoint is received, the byte counter **256** is loaded with the byte count value for that specific endpoint, and the counter is used to count down bytes transmitted to the USB host by the USB RAM device **130**, terminating the packet when the counter reaches zero.

When an OUT token is received by the USB RAM device **130**, the byte counter **256** is zeroed and may be used to count up the number of bytes sent from the USB host to the USB RAM device **130**. The output of the counter is coupled onto the internal data bus **168** (shown in FIG. **8**) so that the byte counts may be written to the dual port RAM device **214** when required.

Endpoint Packet Counter

For each endpoint register, there is a packet counter field **258** stored within the register file **198**. The packet counter field is actually implemented as a bit up/down binary counter. During normal operation, the packet counter for particular endpoint is pre-loaded with the appropriate packet count depending upon which endpoint is being processed and counted down, as each packet is sent to the USB host.

SEQ Bit

The sequence bit which is shown in FIG. **7** as being in bit position **22** is also stored per endpoint register and its bit value distinguishes between DATA0 and DATA1. The bit is zero by default when loaded and it is toggled in place when the appropriate condition occurs. This bit is used to set outgoing data PID and to test incoming data PID's.

Endpoint Enable Bit

Each endpoint register **259** in the endpoint register file **198** shown in FIG. **5** has associated with it an endpoint enable register bit **264** that is 'zero' by default. This enable bit is loaded from the corresponding virtual endpoint register storage location **248** within the dual port RAM device **214** and is therefore set by the microcontroller device **140** and reset by the USB RAM device **130**.

Endpoint Type Field

The endpoint type field **266** is 3 bits wide and specifies the type of endpoint that the endpoint register has been assigned. The default values are implementation specific but at least one endpoint register must always be dedicated to the default control pipe. Beyond this, there are very few constraints on endpoint register assignments and the "interface" or endpoint register assignment is under the control of the

USB RAM device 130 and should match the relevant descriptor. This feature allows a broad flexibility in USB interfaces. The endpoint type field is read by the USB RAM timing and control circuit 174 in order to determine the appropriate behavior for the endpoint. The USB host presumably is aware of the endpoint configuration and thus provides appropriate tokens for such configuration.

Page and Index Pointer Register Field

The page and index pointer register field 268 comprises the remaining 11 bits of each endpoint register and it is dynamically implemented as a pre-loadable binary up/down counter which serves primarily to access data bytes within the dual port RAM device 214 when packets are sent or received by and from the USB RAM device 130. Typically, this pointer field "page plus index" operates as an 11 bit address register, however some implementations will find it convenient to preserve the page value and to "wrap" the index instead of "carry" the same into the page. Thus, the preferred implementation provides an option to support either of these paging methods. This 11 bit pointer register field contains the address of a data buffer within the dual port RAM device 214 that is used for data transfer between the USB host and USB RAM device. Validity Bit

The validity bit 270 is required by the serial interface engines 156 and 154 to determine that a specific endpoint register is valid or not. The microcontroller device 140 does not necessarily use all available USB RAM endpoints. For example, there may be some USB RAM specified endpoints such as the "bulk" that may not ever be used by the microcontroller device and therefore receipt of such an endpoint would be an invalid situation determined by testing the validity bit 270.

It should be noted that each of the endpoint register fields are loadable from the internal data bus 168 shown in FIG. 5.

In FIG. 8, an endpoint register is shown from among the endpoint registers in the endpoint register file 198 of FIG. 5 to show further details of the coupling of each of the endpoint registers to the data bus 168. As shown in FIG. 8, an endpoint register 259 is shown coupled to the internal data bus 168 for transfer of bi-directional information in terms of loading or programming of the endpoint register as well as providing contents of the register. The endpoint register 259 is further capable of being loaded with default values through lines 274 at a time when the USB RAM device 130 is being reset and upon certain conditions occurring. However, default values loaded into the endpoint register can be overwritten and the register can be reloaded at anytime before using the contents of the register.

With respect to the fields in each endpoint register as described in FIG. 7, the endpoint byte counter 256 and the endpoint packet counter 258, as well as the page and index register 268 are all preloadable fields that have up/down counting capability. That is, the byte counter, packet counter and page and index pointer fields are reloadable up/down counters.

The Virtual Endpoint Register File

Referring back to FIG. 5, the endpoint register file 198 is not directly accessible to the microcontroller device 140. Instead, the virtual endpoint register file storage location 248 within the dual port RAM device 214 stores a file of virtual endpoint registers that is directly accessible to the microcontroller device 140.

When the client software has chosen a specific configuration of endpoint registers from the configuration descriptor or other descriptors, the microcontroller device 140 must write the appropriate information into the virtual endpoint register file storage location 248 and then command the USB RAM device 130 to copy that information into the endpoint register file 198. After copying the register information, the USB RAM device 130 is configured and functional.

Because the virtual endpoint register and other registers in the dual port RAM device 214 must be accessible by the USB RAM device 130, an address pointer, stored in the page and index pointer register 268 (shown in FIG. 6), is coupled onto the address pointer bus 212 for accessing data from the dual port RAM device 214 and placing it onto the internal data bus 168, is provided.

As earlier discussed, the microcontroller device 140 is capable of loading data into the virtual endpoint register file storage location 248 for subsequent use by the USB RAM device 130 upon transfer of the data to the endpoint register file 198.

The procedure for the microcontroller device 140 loading data into an endpoint register within the endpoint register file 198 is as follows:

1. The microcontroller device 140 loads data into one or more of the register storage locations to the virtual endpoint register file storage location 248 within the dual port RAM device 214.
2. The microcontroller device 140 then issues a command to the USB RAM device 130 by writing into the command register 144 at address 0x7FE the command value that will be interpreted as "Load EndPt Reg File."
3. The Timing and Counter unit 174 the USB RAM device 130 decodes the command, and, if appropriate, stores the low byte portion of the address of the register storage location of the virtual endpoint register file storage location 248 (loaded in step 1. hereinabove) into the working pointer circuit 206.
4. The USB RAM device 130 couples the contents of the page and index pointer register 268 onto the address pointer bus 212 and reads the low byte address portion referred to in step 3.
5. The low byte portion, read by the USB RAM device 130 in step 4., and addressed by the working pointer circuit 206 is transferred via the internal data bus 168 and latched into the (selected) endpoint register within the endpoint register file 198.
6. The procedure is repeated in a similar manner as described above for the middle and high bytes of the address of the register storage location of the virtual endpoint register file storage location 248 (loaded in step 1. hereinabove) by incrementing the working pointer circuit 206 and repeating steps 4 and 5 for each of the middle and high byte portions.

Error Retry

During the normal course of operation of the USB RAM device 130, the endpoint registers, within the endpoint register file 198, are used to count packets, address data in storage locations within the dual port RAM device 214, and generally support the creation, reception, and error checking of data packets transferred to and from the USB host 128. In some cases, the endpoint register must be returned to the state that existed before the packet was sent, in order to support a "retry" on the part of the host. This will often require that the contents of the page and index pointer register 268 (shown in FIG. 7), the endpoint byte counter 256 (shown in FIG. 7), and the SEQ bit 260 to be preserved. The following discussion details the general operational procedure for supporting such retries.

Recall that each endpoint register 259 within the endpoint register file 198 consists of four bytes containing several fields, two of which comprises the page and index pointer

268, as shown in FIG. 7. Specifically, the low byte of the page and index pointer register 268 contains an index into a page, and generally points to the "next" location in dual port RAM device 214 from which a byte will be read or to which a received data byte will be written. In order to be able to retry a specific transfer, it is often necessary to temporarily save the index portion of the contents of the register 268, the SEQ bit 260; and the packet counter 258. In an embodiment of the present invention, the index is saved in the virtual endpoint register where it may be reloaded if a retry becomes necessary. The saved virtual endpoint register index is updated only if the transfer is successful. The same general approach is applied to the SEQ bit, and the packet counter. Both of these states are maintained in the endpoint register 259. The update of these fields is delayed until an ACK is received from the USB host, or until all checks are completed on incoming packets. If the packet transfer fails, the updates are inhibited. When the outcome of the packet transfer is detected as being successful, the SEQ bit is toggled, the packet counter 258 is decremented, and the index is copied to the index portion of the page and index pointer register 268 within the virtual endpoint register file storage location 248.

Segmentation for Protection

The page-based endpoint register feature of the USB RAM device 130 provides for primary protection and isolation of one endpoint from another and generally partitioning these endpoints such that a stall or problem on one endpoint typically prevents affects on other endpoints.

USB-RAM "Auto NAK" Capability

The receipt of a token for a specific endpoint always initiates a new transaction, and causes the virtual endpoint register in the virtual endpoint register file storage location 248 that is associated with the specific endpoint to be partially loaded into the corresponding endpoint register file 198 thereby pre-configuring the USB RAM device 130 for the transaction. Some transactions are periodic, or in some way predictable, and the pre-configured endpoint registers generally are capable of immediate service in these cases. For example, the ISO IN registers (shown in the third row, or row '2', of the endpoint register file 198) can be preset to point to the data buffer 254 that in FIG. 5 will periodically be sent to the USB host. Some transactions are a periodic and asynchronous, and cannot generally be anticipated, an example of such types of transactions are SETUP transactions on the CONTROL pipe that issue standard request packets to the microcontroller device 140. In most cases the microcontroller device 140 cannot retrieve the request and setup the response thereto in the allowed response time. Therefore, the USB RAM device 130 initiates "auto-NAK" transactions. That is, when the microcontroller device is not prepared to respond to a request from the host, the USB RAM device upon detection thereof, automatically responds to the host with a "NAK" token thereby informing the host that the microcomputer is not ready, which implies that the host must try again later. "NAK" tokens are repeatedly and indefinitely sent to the host by the USB RAM device until such time as when the microcomputer device generates an actual response to the host and signals the USB RAM device that the response is ready. In the preferred implementation this signaling is via interrupt from the microcontroller device.

On an INTERRUPT pipe, the USB RAM device sees an "IN" token every frame or latency period. If the INTERRUPT endpoint is not enabled, the USB RAM device issues NAK to indicate that it is busy.

If the microcontroller device has command/event (incoming call, connect, disc, etc.), then the microcontroller device issues DATA0 on the interrupt pipe in response to "IN" command. The host issues an "ACK" if "DATA0" was sent error free and if an error had occurred, no response is sent.

Let's examine the microcontroller device procedure associated with the SETUP Descriptor Request operation: The USB RAM device recognizes the SETUP Packet ID, checks the address, the endpoint and the CRC. If these are correct the SETUP is recorded as the "last" packet and the EOP is checked. The host then sends an 8 byte long DATA0 packet containing a standard request for a descriptor. The USB RAM device sees the CONTROL endpoint and the "last=SETUP" and tests whether the CONTROL register has been enabled. The USB RAM device enables the (de-stuffed) incoming data buffer onto the internal data bus 168; then uses the CONTROL address and generates a write strobe to the USB RAM device, increments the index, decrements a byte counter and loops until the byte counter reaches zero or an EOP (end-of-packet) is seen. CRC-16 is checked for validity and if valid, the USB RAM device writes ACK into the serial transmit subsystem 156 in FIG. 5, and also sends the microcontroller device a SETUP interrupt by writing into 142 in FIG. 5.

Default Map of Dual Port RAM for Microcontroller Application

At this point, it suffices to briefly discuss the organization of data within the dual port RAM device 214. Referring now to FIG. 9, a memory map 300, which is the default memory map (or organization) of information stored in the dual port RAM device 214. It should be noted that the default memory map 300 is designed to support the Cy123 ISDN Controller device, disclosed in U.S. Pat. No. 5,541,930.

The request storage location 142, for storing requests received from the USB host, is accessed by the address value 0x7FF. When location 142 is written to, an interrupt is generated to the microcontroller device 140 through the "INT" line of the microcontroller lines 216 (shown in FIG. 5).

The command storage location 144, for storing commands received from the microcontroller device 140, is accessed by the address value '0x7FE'. When written to, an interrupt is generated on the 'INT' line of the bus 218 to the circuit 174.

The mailbox storage location is used for storing pointers and comprises of three storage locations within the dual port RAM device 214, each of which is: a mailbox high storage location 301, by '0x7FC' for storing the 8 most significant bits (or MSB byte) of the mailbox information; a mailbox medium storage location 302, addressed by the value '0x7FB', for storing the 8 middle bits of the mailbox information; and a mailbox low storage location 304, addressed by the value '0x7FA', for storing the 8 least significant bits of the mailbox information.

An interrupt address space 306 is assigned for storing interrupt packets starting at address '0x7C0'. The virtual endpoint register file storage location 248 starts at address location '0x780' and the SETUP storage location 250 starts at address location '0x740'.

The USB RAM device is designed to allow default operation with minimal intervention by the microcontroller device. Most applications will probably be able to live with the default memory map 300, but can always over-write any endpoint register as appropriate. Any USB-RAM memory NOT defined for USB transfers is available for use by the application device, that is, the microcontroller 140.

All Bulk operations are page-based and wrap around the page. The microcontroller device is interrupted when valid

data is received or transmitted, and is responsible to prevent overruns or underruns. The microcontroller device can issue NAK's if necessary to throtle the USB host.

Bulk-IN

Referring now to FIG. 10, a Bulk endpoint register storage location 310 is shown included within the endpoint register file 198. The Bulk endpoint register storage location 310 includes a Bulk page register 312 and a bulk index register 314, and a byte count field 316.

When a Bulk <IN> token is received, a Bulk SEQ bit 320 is tested, and DATA0 or DATA1 is sent to the host, followed by ByteCnt bytes of data from the Bulk buffer, defined and accessed via the Bulk-ptr consisting of the address in 312 and 314.

If the data is received by the host with no error, the host sends an ACK to the device 130. When the device sees the ACK, it will toggle the SEQ bit 320, and store a Bulk index field 314 in the corresponding register 318 within the virtual endpoint register storage subsystem 248, indicating the location of the next BULK data byte to send in response to the next <IN>. If no ACK is received, the host detected an error, so the SEQ bit 320 is untouched, and the virtual Bulk index is read and copied into the Bulk endpoint register 310, pointing just past the last acknowledged good data in the Bulk buffer. Successful transfers cause the microcontroller device 140 to be interrupted. The microcontroller device will then use the index field 318 to determine how much data has been sent, to prevent overrun. The USB RAM device responds to <IN> tokens until the Bulk packet counter 258 (FIG. 6) counts down to zero.

Bulk-OUT

Referring now to FIG. 11, a Bulk-Out endpoint register storage location 326 is shown included within the endpoint register file 198. When a good Bulk data packet is received by the USB RAM device from the host, the index is saved in the virtual Bulk-Out index field 322 of the virtual endpoint register file storage location 248. The SEQ bit 324 pertaining to the Bulk-Out endpoint register of the endpoint register file 198 is toggled, and an ACK is sent to the host. Optionally, the microcontroller device is interrupted. If bad data is received from the host, the Bulk-Out index field in 326 is loaded from the virtual endpoint register file storage location 322, that is, the next location past the last good data is recovered, no ACK is sent to the host, and the SEQ bit 324 is untouched. The host will detect the lack of an ACK and will retry the OUT Data transaction. The DATA0/1 sequence is handled by the host. DATA0 is chosen when the Bulk pipe is configured and the Bulk endpoint register storage location 326 is loaded. If the Bulk pipe is stalled, both the USB host and the USB RAM device should reset to DATA0 when STALL is cleared. Bulk transactions are re-tried if errors are detected. Successful transfers cause the microcontroller device to be interrupted. The microcontroller device will use the virtual index field 322 index to determine how much data has been received.

Interrupt I/O

As shown in FIG. 12, the value in the default INTERRUPT endpoint register 332 of the endpoint register file 198 is stored in and accessed from the interrupt address space 306, at a location addressed by the value '0x7C0', within the dual port RAM 214. A maximum packet size, in terms of bytes, of 64, is assigned to the this endpoint, as indicated by byte count 328. If the INTERRUPT endpoint is disabled, the USB RAM device will respond to all Interrupt <IN> tokens with 'NAK'. The default interrupt latency is 1 millisecond. When INTERRUPT is enabled (by the microcontroller device), the USB RAM issues a DATA0 to the host and then reads 64-byte packet of data from locations 0x7C0 to 0x7FF within the dual port RAM and sends it to the host. A default packet count 330 is one, but the microcontroller device can select larger packet counts if appropriate.

In FIG. 12, the byte count 328 and the default packet count 330 are stored in an interrupt register 332 within the endpoint register file 198, which also includes a page pointer 334 having the value '7' and an index pointer 336 having a value that is within the range of the addresses assigned for storing interrupt information in the dual port RAM device, i.e. C0-FF (in hexadecimal notation).

ISO-IN

Referring now to FIG. 13, the value of the default ISO-IN endpoint register 350 of the endpoint register file 198, is configured to point to the beginning of a 256-byte page, as shown in FIG. 13 at 352 (a page of storage location is shown at 354) and the bandwidth constraints are optimized by choosing four packets of 64 bytes each. This will cause the page to be sent in 4 milliseconds, while in ISDN applications, bandwidths cause a page to be filled every 16 msec. Thus, the B-channel (used in ISDN applications) double buffers will be filled in 16 msec and drained in 4 msec. The microcontroller device must swap pages within the dual port RAM 214 before issuing the next ISO-IN-buf_ready command to USB RAM device.

ISO-IN packets are always DATA0 and are unacknowledged. No retries occur in isochronous pipes. If the packets have been transmitted, the USB RAM device will issue a 'NAK' for each ISO-IN <IN> token. The next microcontroller device ISO-IN command to the USB RAM device will cause the ISO-IN endpoint register to be reloaded, and the endpoint will be re-enabled by setting the enable bit 364 of FIG. 7. It should be noted that isochronous pipes never STALL (since there is no handshake).

ISO-OUT

In FIG. 14, the default ISO-OUT end point register 356 is shown as being configured to point to the beginning of a 256 byte out-page 358 in the dual port RAM 214. This is to be part of a double buffered pair of pages. By default, the 256 page will receive four 64 byte packets in four milliseconds. This is controlled by the client software on the host side. Typically, the ISO-OUT transfers will be initiated by an IRP "Interrupt Request Packet" from the client, in response to an interrupt from the device. For ISDN B-channel data, this assumes approximately one msec for the 'buffer_rdy' interrupt, and four msec to fill the buffer. Since buffers are swapped every 16 msec, this 5 msec transaction is OK, that is, the Data_Out_buffer will be filled in 5 msec, while it takes 16 msec to drain.

ISO-OUT packets are always DATA0 and are unacknowledged. It is assumed that the client software IRP's will issue the correct commands to the host software such that only four 64 byte packets will be sent per B-channel interrupt.

Control

Control transfers are intended to support configuration/command/status type bidirectional communications between the client software and the device. As shown in FIG. 15, control transfers consist of a SETUP packet 360 from the USB host 128 to the USB RAM device 130, followed by either no data transactions or one or more data transactions 361 in the setup specified direction, and a STATUS packet 362, which is transferred from the USB RAM device 130 to the USB host 128: the SETUP packet is 8 bytes long and has a USB-specified structure. Data transactions following SETUP have no USB-defined structure, but will usually have a user-defined structure.

The status transaction returns "success" when the endpoint has completed processing the requested operation. The

host can advance to the next CONTROL transfer after status is returned. The only USB defined "default" pipe is a CONTROL pipe with endpoint 0. This is the pipe used to configure the system. Additional CONTROL pipes can be defined. CONTROL pipes offer "best effort" delivery. CONTROL pipes can have a maximum packet size of 8, 16, 32, or 64 bytes. The maximum packet size is always used for data payloads.

USB RAM Device Auto-NAK's on OUT Transfers

Note that the host can transmit any number of <OUT> DATA transfers. This represents a potential problem since these data transfers may arrive at the USB RAM device faster than the microcontroller device can handle them. After the first <OUT> has been transmitted by the host, the microcontroller device can NAK the host, thereby preventing following data transfers from occurring immediately. That is, <OUT> packets are not processed and responded to by the USB RAM device sending 'NAK' unless the endpoint is enabled. In this way, the microcontroller device can reset the CONTROL pointer to an alternative location. Note that although CONTROL packets also wrap around a page, the default location for the CONTROL pointer is 0x740, and wrapping around page 7 will typically lead to problems. Therefore, in the default case, the user software should limit the amount of data sent to the device via CONTROL packets, and/or the USB RAM device should throttle the host via NAKs. The host will retry NAKed transactions at a later time.

1.) If a new SETUP is received before an old control transfer is completed, abort old transfer and handle new SETUP.

2.) a stalled CONTROL endpoint should still accept SETUP PID.

USB RAM Device Page-based Endpoints

All USB RAM device transfers are page-based. That is, the USB RAM device includes memory that is divided into 8 pages of 256 bytes (0x100) each. When data transfer reaches the last byte in a page, 0xNFF, the pointer will 'wrap' around to the first byte on the page, 0xN00, instead of advancing to the first byte on the next page, 0x(N+1)00. This limits the damage that errors on one endpoint can have on other endpoints.

USB RAM Device Commands

There are two classes of USB RAM device commands:

1. Generic commands
2. pass thru commands

List of USB RAM Device Commands

Generic Commands:

'0' Load Endpoint Reg #0 (in endpoint register file 198 from Virtual Endpoint Reg. #0 (in virtual register file storage location 248) and enable

'1' Load Endpoint Reg #1 from Virtual Endpoint Reg #1 and enable

'2' Load Endpoint Reg #2 from Virtual Endpoint Reg #2 and enable

'9' Load Endpoint Reg #9 from Virtual Endpoint Reg #9 and enable

':' set the Control_enable and load Control_index

';' reserved

'<' reserved

'=' copy host assigned address into device address register

'>' reserved

'?' dump Endpoint Reg file to dpRAM using pointer at 0x7FA,0x7FB

Pass Thru Commands:

'all other codes' setup INTERRUPT endpoint register in virtual EndPt register file 248, (FIG. 5) and set Interrupt_enable bit of this EndPt register.

Discussion of the Generic Commands

Generic commands are coded as 0x3X (i.e. '0' . . . '9', ':', ';', '<', '=', '>', '?') and are generally application and configuration independent. The ASCII decimal digit commands specify the endpoint register to be loaded from the virtual endpoint register file 248, for example, command '2' signals USB RAM device to load endpoint register #2 from #2 in the virtual endpoint register file, where the default control pipe is always endpoint register #0.

Command '=' (0x3D) advises the USB RAM device that the host-assigned device address is available, and should be copied from the dual port RAM device 214 into the address storage location 146.

Command ':' (0x3A) informs the USB RAM device that a SETUP packet has been received, and the USB RAM device should load the control endpoint register with the default index 0x40 and set the CONTROL_enable bit, that is, the Enable bit 264 (FIG. 7) of the control EndPt register.

Discussion of Pass Through Commands

All other commands discussed above are to be passed through the USB RAM device to the host, and through the host to the client (or user). Although the user software may be written to support other configurations, the recommended procedure for "pass-through" commands is as follows:

The microcontroller device (or application device) issues a command to the USB RAM device by writing to the command storage location 144 (at address 0x7FE in the dual port RAM device 214.) The USB RAM device 130 attempts to interpret the command. If the command is a generic command, then no assumption is made about the endpoint configuration other than assuming that the default control pipe is endpoint 0. If it is not a generic command code, or '!', then the USB RAM device assumes that it is a command code to be passed through to the host, and therefore an INTERRUPT endpoint exists, and further that the INTERRUPT endpoint is endpoint #1.

The USB RAM device then copies the Virtual Endpoint Reg #1 into Endpoint Register #1, and sets INTERRUPT_enable. By default, Endpoint #1 is setup to send one 64-byte packet of data, the data to be read from the dual port RAM device 214, at addresses 0x7C0 . . . 0x7FF. After the INTERRUPT packet is sent, the USB RAM device clears the INTERRUPT_BUSY state by writing zero to 0x7FE.

The ISO_IN_enable is not used, but could be, in addition to the PktCntr[ISO_N]

A generic command loads all of the virtual endpoint registers' information from the virtual endpoint register file storage location 248 into corresponding register locations in the endpoint register file 198, thus the corresponding SEQ bit must be current. Everywhere the SEQ bit is toggled, its image in the appropriate virtual endpoint register file storage location 248 must be updated.

The Enable and SEQ bits operate as follows. Generally, the USB RAM device will toggle the SEQ bit (and zipdate its Virtual image) while the microcontroller device sets the Endpoint_enable bit and the USB RAM device resets it.

Example 8051 (microcontroller device 140) response to CONTROL SETUP command from the USB RAM Device:

Upon the detection of an interrupt from the USB RAM device, the 8051 (the 8051 is a commercially available microcontroller device from Intel and it is used as an example of the microcontroller device) reads the command storage location 142 (at address location 0x7FF in the dual port RAM device 214) and retrieves the 'SETUP' code, which informs the 8051 that the 8-byte SETUP data has been stored in locations 0x740 . . . 0x747 of the dual port RAM device 214. The 8051 then interprets this data and deter-

mines that the standard request packet is a Set_Address command from the USB host. The 8051 will then command the USB RAM device by writing '=' to the command storage location 144 of the dual port RAM device 214, at location 0x7FE. The USB RAM device reads the command storage location 144, determines whether it is the 'Address='-command, reads the contents of the location 0x744 onto its internal data bus and writes this address into its address storage location 146. All incoming tokens with this address will now be recognized by the USB RAM device. Prior to this, only the default CONTROL 0 pipe was recognized.

If the standard request packet is any other Set_xxx command, then the application device is assumed to know what to do with the command. Similarly, any Get_xxx request packet from the host must be interpreted by the application microcontroller device, and the appropriate data sent back to the host, transparent to the USB RAM device.

More specifically, if the standard request packet is any Get_xxx request, the application device should setup the desired information at an appropriate location (by default, starting at the CONTROL base, 0x740 . . . 0x780) and then command the USB RAM device by writing the Setup code, ':', into the command storage location 144. This assumes that the USB RAM device has been NAK'ing the <IN> tokens from the host, since the CONTROL-enable bit is assumed reset. The ':' command is read by the USB RAM device and causes the USB RAM device to load the CONTROL endpoint register with the default CONTROL base (0x740), enable the endpoint, and respond to the next <IN> token by sending a Data1 packet using the CONTROL register parameters.

When the 8051 (microcontroller device) sees the 'setup' command from the USB RAM device, it will read the request packet from locations 0x740 to 0x747. If the host sends more data as <OUT> packets, the USB RAM device writes the data starting at the CONTROL base at location 0x740. If, instead, data is requested from the 8051, the 8051 writes the data into USB RAM device, starting, at the CONTROL, base, and issues the ':' command to the USB RAM device. The USB RAM device will respond to the next <IN> token on the CONTROL pipe by sending a Data1 packet. If the 8051 sees a vendor specific request packet, with no following data required, then the microcontroller device should respond with a zero-length Data1 packet for the Status stage of the setup.

USB-to-ISDN Layered Architecture and Time-Multiplexed Pipes

To this point, a general purpose USB-to-microcontroller interface has been described. Hereinafter, an optimal solution is presented for interfacing the USB host to the world-wide Integrated Services Devices Network (ISDN) network through an ISDN adapter. In FIG. 16, a high level diagram is shown to include two major interfaces: a USB interface 400 for interlacing an electronic communication device such as a PC 402 to an ISDN adapter 404; and an ISDN interface 406 for interfacing the adapter device 404 through an ISDN communication link 408 to various types of communications devices (not shown). In an example embodiment of the present invention, the ISDN adaptor 404 may be implemented as the USB RAM device 130, and a Cybernetic Micro Systems, Inc. CY123 device. Furthermore, the PC 402 is an example embodiment of the USB host 128. Because both USB and ISDN are layered architectures, and because both use time-multiplexed communication channels, the actual interfaces involved are shown in FIG. 17.

Layered architecture is discussed in U.S. Pat. No. 5,541,930, the disclosure of which is incorporated herein by reference, as are time multiplexed pipes. In FIG. 17, the layered architecture of ISDN is shown, through a dotted line at 410, to extend through the layered architecture of USB. Additionally, a mapping is shown at 412 between ISDN time multiplexed 'channels' 414 and USB time multiplexed 'pipes' 416. The 'channels' 414 form the communication link 408 (shown in FIG. 16) and comprise of a B1 channel 418, a B2 channel 420, a D channel 422 and an EOC/M channel 424. Each of these channel is described in greater detail in the above-referenced and incorporated patent application. The 'pipes' 416 are as described above with reference to figures preceding FIG. 16.

In FIG. 17, a client (or user) 424 is shown to be communicating with the USB host 128, through network layers of the USB host: a function layer 426; followed by USB device layer 428; followed by a USB bus interface layer, to the USB RAM device 130 through the USB interface 400.

This communication extends through the USB RAM device 130 by going through similar layers in the order shown by the dotted line 410 and thereafter continues, through the ISDN interface 406, to the ISDN communication link 408 layers: a physical layer 432; a data link layer 434; and a network layer 436.

As described in greater detail in the above-referenced and incorporated patent document, an interrupting dual port RAM provides a powerful interface element capable of supporting both MESSAGE and STREAM communications. The ISDN D channel 422 is MESSAGE-based, while the B1 and B2 channels are primarily STREAM-based. The USB CONTROL pipe, within the pipes 416, is primarily MESSAGE-based while the ISOCHRONOUS and BULK pipes, within the pipes 416, are STREAM-based. Therefore, in principle, it should be possible to map MESSAGE-based ISDN communications into MESSAGE-based USB, and similarly, STREAM based ISDN into STREAM based USB, and vice versa and the present invention effects such mappings in a flexible general purpose architecture subject to the previously discussed constraints of non-DMA type microcomputers.

The MESSAGE mode is the only bi-directional mode available. Messages transfer using a CONTROL pipe. STREAM mode is for uni-directional DATA transfers, and applies to the INTERRUPT, ISOCHRONOUS, and BULK pipes. Thus, the host may issue Layer 3 ISDN commands over a CONTROL pipe, while B-channel data may flow over ISOCHRONOUS pipes.

Although the present invention has been described in terms of specific embodiments it is anticipated that alterations and modifications thereof will no doubt become apparent to those skilled in the art. It is therefore intended that the following claims be interpreted as covering all such alterations and modification as fall within the true spirit and scope of the invention.

What I claim is:

1. A RAM-based interrupt-driven interface device for establishing a communication link between a high performance serial bus host and a microcontroller device for providing a control function, the interface being operative to receive digital information in the form of command, data and control packets from the host and to process the packets and communicate the processed digital information to the microcontroller device, and in response thereto, the microcontroller device being operative to communicate digital information to the interface device for processing and transfer to the host, comprising:

means for receiving through said serial bus, a command generated by the host;